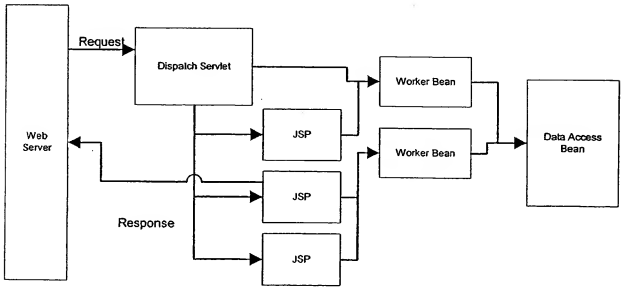
Application Management: This component's responsibility is to provide application management interface. This component is accessible for updates only by the ASP administrator. A detailed list of interfaces for this component is described in the interfaces section.

License Manager: This component's responsibility is to manage the licenses. SLiM server will check out licenses from the license manager. A detailed list of interfaces for this component is described in the interfaces section.

The architecture for the Web Server extensions implementation is shown below:



The basic elements of this architecture are as follows:

1. Every request into the system goes through a dispatcher servlet. This servlet will perform initialization, initial validation of the request and miscellaneous checks before dispatching the request to a JSP page. A worker bean will responsible for performing the initialization. The processing of the incoming request is performed at this stage. The request is then dispatched to an appropriate JSP page.
2. The JSP page will invoke worker beans to access the dynamic data from the database via the Data Access Bean and the resultant page is sent back to the user.

This architecture is illustrated with the following example.

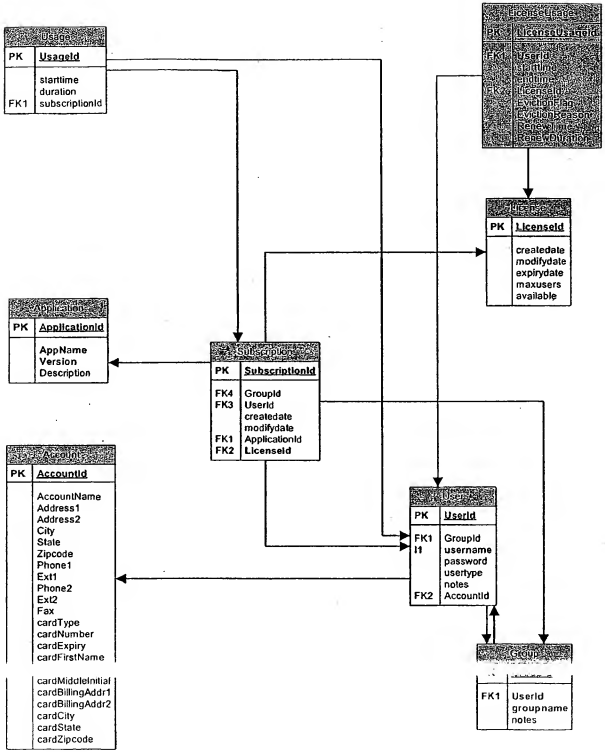
1. User sends in a request to update the username and password information in the database. Inputs are username, old password, new password.
2. The dispatch bean will call the user(worker) bean to:
    a. Validate the user's old password.
        i. The user worker bean will make a request to the data access bean to access the password for the user.
        ii. The two passwords are compared and the result is returned.
    b. If the password was valid then, update the new password.
        i. Call the data access bean to update the password in the database.
    c. Else return failure.
3. Based on the success or failure the dispatcher will dispatch the page request to the appropriate JSP page. (eg. error.jsp on failure and user.jsp on success).
4. The page will invoke the appropriate the worker bean (error bean or user bean) to obtain the dynamic data and send the response back to the user.

The salient features of this architecture are:

1. Presentation and processing logic is separate. Thus, the customer(ASP) can customize the look and the feel of the pages without impacting the processing logic as it is segregated.
2. The data access bean is separated from the worker beans, which are primarily responsible for the business logic. This allows us to change the data access layer (eg enabling LDAP access) in the future without impacting the system drastically.

Data type definitions

The central data structure for Web Server is the database model. The overall database model for user and subscription management is shown below.


Usage
PK Usageld
starttime
duration
FK1 subscriptionId
PK Licenseld
createdate
modifydate
expirydate
maxusers
available
Application
PK ApplicationId
AppName
Version
Description
Subscription
PK SubscriptionId
FK4 GroupId
FK3 UserId
createdate
modifydate
FK1 ApplicationId
FK2 Licenseld
Account
PK Accountid
AccountName
Address1
Address2
City
State
Zipcode
Phone1
Ext1
Phone2
Ext2
Fax
cardType
cardNumber
cardExpiry
cardFirstName
cardMiddleInitial
cardBillingAddr1
cardBillingAddr2
cardCity
cardState
cardZipcode
PK Userid
FK1 GroupId
I1 username
password
usertype
notes
FK2 Accountid
FK1 Userid
groupname
notes

The important features of this data model are:

Account: Table holding all the billing and contact information for a user or a group.

User: An end user in the system. A user can optionally belong to a group.

Group: A group of users. One of the users in the group is designated as the group administrator. Each group has a unique account associated with it.

Application: This table contains the data about various applications in the supported by the ASP.

License: Each row in this table corresponds to the licensing term for a given subscription. This table also maintains the active count of the licenses checked out.

Subscription: This table contains entries for subscription items. A subscription item consists of user/group, application and license.

Usage: This table contains the runtime information for a system. SLiM server updates this table with access token usage data. A billing system may interface with this table to generate billing data. A reporting system may interface with this table to report on usage patterns.

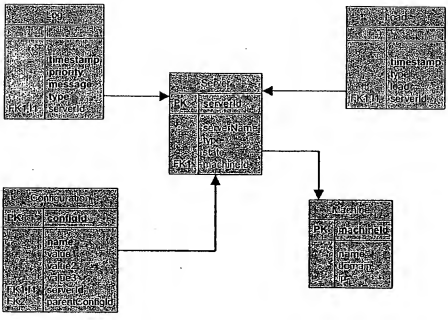
LicenseUsage: This table is responsible for recording checked out licenses in the system.

The data model for storing the server related information is shown below:

PK: Primary Key for the table.

FK: Foreign Key. Used for relations between tables.

I1,I2.. : Index Columns.



The tables in this model are:

Server: This table contains entries for each logical server in the system.

Machine: This table contains entries for each physical server in the system

Configuration: This table contains configuration entries for a given server. The configuration entries can be hierarchical in nature. Each configuration has the following format:

Name Value1 [Value2] [Value3] [ParentConfigId]

Load: This table maintains the historical and real-time load information for a given logical server in the system.

Log: This table maintains the logs for a logical server in the system. The log messages saved here are "major" events in the logical server system. A detailed logs stored in a flat file on the physical machine containing the logical servers.

Global Data Structures:

```
struct ServerTuple
{
int serverId,
int type,
String serverName
};
```

```
struct Couple
{
String name,
String value
};
```

For the Access Token and related data structures, please refer to the SLiM server Low Level Design Document. The interfaces below will discuss some of the API's based on the these data structures.

Interface definitions

The interfaces exposed by various sub-components are detailed below.

Server Management Component:

CreateServer

int CreateServer (ServerConfig* config)

```
Input:
  Server Configuration.
Output:
  Unique ID for the server.
Comments:
  Create a logical server with predefined con-
figuration.
Errors:
  INVALID SERVER ID
  NO DATABASE CONNECTION
  UNKNOWN SQL ERROR
```

UpdateServerConfig

Bool UpdateServerConfig(int serverId, String name, String value)

```
Input:
  Server Id
  Config name and value
Output:
  Unique ID for the server.
Comments:
  Create a logical server with predefined con-
figuration.
```

```
Errors:
   INVALID SERVER ID
   NO DATABASE CONNECTION
   UNKNOWN SQL ERROR
```

AddMachine

Bool AddMachine(String name, String domain, String ip)

```
Input:
   Machine name,domain and ip.
Output:
   Success/Failure
Comments:
   Create a physical machine entry.
Errors:
   NO DATABASE CONNECTION
   UNKNOWN SQL ERROR
```

SetServerLog

Bool SetServerLog(int serverId, LogTuple log)

```
Input:
   Server Id
   Log tuple(data structure in the Logging
document.)
Output:
   Success/Failure
Comments:
   Add the log data for a server
Errors:
   INVALID SERVER ID
   NO DATABASE CONNECTION
   UNKNOWN SQL ERROR
```

GetServerLog

LogTuple[] GetServerLog(int serverId, int maxrows = 25)

```
Input:
   Server Id
   Maxrows: Maximum number of rows to be re-
turned.
```

```
Output:
   Array of Log tuples(data structure in the
Logging document.)
Comments:
   Get the log data for a server
Errors:
   INVALID SERVER ID
   NO DATABASE CONNECTION
   UNKNOWN SQL ERROR
```

GetServers

Server[] GetServers()

```
Input:

Output:
   Array of Server tuples(data structure de-
fined above)
Comments:
   Get all the server information
Errors:
   INVALID SERVER ID
   NO DATABASE CONNECTION
   UNKNOWN SQL ERROR
```

SetServerState

Bool SetServerState (int serverId, short state)

```
Input:
   ServerId: Unique id for a server
   State: State information for a server.
Output:
   Bool True/False for success/failure.
Comments:
   Update the database with current state in-

Errors:
   INVALID SERVER ID
   DB ROW LOCKED
   NO DATABASE CONNECTION
   UNKNOWN SQL ERROR
```

GetServerState: Obtain the last known state for a specified server

short GetServerState (int serverId)

```
Input:
  ServerId: Unique id for a server
Output:
  State: State information for a server.
Comments:
  Obtain the last known state for a specified
server
Errors:
  INVALID SERVER ID
  DB ROW LOCKED
  NO DATABASE CONNECTION
  UNKNOWN SQL ERROR
```

GetServerConfig: Obtain configuration information for a specified server

ServerConfig* GetServerConfig (int serverId)

```
Input:
  ServerId: Unique id for a server
Output:
  ServerConfig*: State information for a
server. (ServerConfig data structure is defined
in the server configuration document).
Comments:
  Obtain the last known state for a specified
server
Errors:
  INVALID SERVER ID
  DB ROW LOCKED
  NO DATABASE CONNECTION
  UNKNOWN SQL ERROR
```

SetLoadData:

void SetLoadData (int serverId, int Load)

```
Input:
  ServerId: Unique id for a server
  Load: Load for the server
```

```
Output:

Comments:
   Monitor may call this interface to persis-
tently store historical load data.  It is still
not clear if SLM and application servers will
store this directly themselves.
Errors:
   INVALID SERVER ID
   NO DATABASE CONNECTION
   UNKNOWN SQL ERROR
```

GetServerLoad:

void GetServerLoad (int serverId, , int maxrows = 25, int** Load)

```
Input:
   ServerId: Unique id for a server
   maxrows: Maximum number of rows to be re-
turned.Default is 25.
Output:
   Load: Load for the server
Comments:
   Obtain server component load information to
manage load balance.
Errors:
   INVALID SERVER ID
   NO DATABASE CONNECTION
   UNKNOWN SQL ERROR
```

FlushLoadData:

void FlushLoadData (<tuples> LoadData)

```
   LoadData tuples containing <server id,
server load> values.
Output:

Comments:
   Used to flush aggregated load data to the
databa
```

```
Errors:
   INVALID SERVER ID
   NO DATABASE CONNECTION
   UNKNOWN SQL ERROR
```

User/Account/Subscription Management Component

CreateUser. This API is used to create user record in the system. Arguments will be Username, Password.

Bool CreateUser(String username, String password)

ValidateUser. This API is used to validate user record in the system. Arguments will be Username, Password.

Bool ValidateUser(String username, String password)

CreateAccount. This API is used to create account records in the system. Arguments will be billing address, credit card information etc.

Bool CreateAccount(String username, <Account Information>couple[])
Input:
Username associated with the account.
An array of names and values for the account.

AddSubscription. This API is used by the end users/group administrators to subscribe to applications.

Bool AddSubscription(<Subscription Information>couple[])
Input: An array of names and values for the subscription.

UpdatePassword. Used to change user information. Password, username etc.

Bool UpdatePassword(String username, String old-password, String new-password);

UpdateAccount. Used to update the account information. Billing Address etc.

Bool UpdateAccount(Couple[])

Input: An array of name, value pairs for the fields to be updated.

UpdateSubscription. Used to add additional time to a subscription.

Bool UpdateAccount(Couple[])

Input: An array of name, value pairs for the fields to be updated.

GetUserRecord. Used to get current user configuration.

Couple[] GetUserRecord (String username)

GetAccountRecord. Used to get current account configuration for a user.

Couple[] GetAccountRecord(String username)

GetSubscriptionRecords. Used to get to subscription records in a database. End user may just want to verify what they are subscribed to.

Couple[][] GetSubscriptionRecords(String username)

Output: An array of array of couples containing the subscription information for a given user.

DeleteUser. Used to delete users who are no longer valid in the system. Typically called by the ASP admin.

Bool DeleteUser(String username)

DeleteAccount. Used to delete un-used accounts.

Bool DeleteAccount(int accountId)

DeleteSubscription. Used by the ASP admin to remove subscriptions.

Bool DeleteSubscription(int subscriptionId)

Group Management Component

CreateGroup. This API is responsible for creating group accounts in the database. Called by the group admin user.

Bool CreateGroup(String groupName, String admin, String notes)

AddUserToGroup. Adds a user to a group.

Bool AddUserToGroup(String groupName, String username)

DeleteUserFromGroup. Removes a user from a group.

Bool DeleteUserFromGroup(String groupName, String username)

GetActiveSessions. Gets the active sessions for a group.

Couple[][] GetActiveSessions(String groupName)

Output: An array of array of couples containing the following information for each active session in the system:

Username
LicenseId
StartTime
EndTime
Subscription

Licensing Component

CheckoutLicense: Checks out a license.

int CheckOutLicense(int subscriptionId, long* pStartTime, long* pStopTime)

Inputs:

SubscriptionId: Subscription id of the user.

Outputs:

>0 for a successful license. The output is the license usage id.
StartTime for the license.
StopTime for the license.

Comments:

The system should validate the availability of the license.

Errors:

```
INVALID USER ID
INVALID SUBSCRIPTION
LICENSE NOT AVAILABLE
NO DATABASE CONNECTION
UNKNOWN SQL ERROR
```

RefreshLicense: Refreshes a license.

int RefreshLicense(int LicenseUsageId, long* pStartTime, long* pStopTime)

Inputs:

LicenseUsageId: License usage id.

Outputs:

>0 for a successful license. The output is the license usage id.
StartTime for the license.
StopTime for the license.

Comments:

The system should validate the availability of the license.

Errors:

```
INVALID USER ID
INVALID SUBSCRIPTION
LICENSE NOT AVAILABLE
NO DATABASE CONNECTION
UNKNOWN SQL ERROR
EVICTION
```

CheckinLicense: Check in a license

Bool CheckInLicense(String username, int subscriptionId, int licenseUsageId)

Inputs:

Username: user trying to check out the license
SubscriptionId: Subscription id of the user.
LicenseUsageId: Usage id for the checked out license.

Outputs:

Success/Failure

Comments:

Errors:

```
INVALID SUBSCRIPTION
NO DATABASE CONNECTION
UNKNOWN SQL ERROR
```

ValidateLicense: Validate that the user has a license checked out.

Bool ValidateLicense(String username, int subscriptionId, int licenseUsageId)

Inputs:

Username: user trying to check out the license
SubscriptionId: Subscription id of the user.
LicenseUsageId: Usage id for the checked out license.

Outputs:

Yes/No.

Comments:

Errors:

```
INVALID USER
INVALID SUBSCRIPTION
INVALID LICENSE
NO DATABASE CONNECTION
UNKNOWN SQL ERROR
```

DBAcquireAccessToken

RPCReturnCodes DBAcquireAccessToken(long SubscriptionId, long* pAccessTokenId, string UserName, string Password, long* pStartTime, long* pStopTime, long* ApplicationId)

IN	SubscriptionId	Id of the subscription being used.
IN/OUT	pAccessTokenId	-1 if this is a first time access.
IN	UserName	Username string.
IN	PassWord	Encrypted Password
OUT	pStartTime	Start time for Access Token validity.
OUT	pStopTime	Stop time for Access Token validity.
IN/OUT	ApplicationId	Id of the application. –1 Default.
OUT	RPCReturnCodes	RPC Return codes.

Processing:

This is fairly complex function. The processing involved in this function call is:

- If this is the first access (ie *pAccessTokenId == -1) then **ValidateUser**
- If the ApplicationId is –1 then **GetAppId**
- If this is the first access (ie *pAccessTokenId == -1) then **CheckoutLicense**
- If this is a renewal request: **RefreshLicense**
- If there is a failure and it is due to eviction: **GetEvictionReason**

Errors:

```
#define     RPCR_USER_AUTH_FAILED
#define     RPCR_ACCESS_TOKEN_INVALID
#define     RPCR_ACCESS_TOKEN_EXPIRED
#define     RPCR_LICENSE_NOT_AVAILABLE
#define     RPCR_LICENSE_ALREADY_HELD
#define     RPCR_EVICTION_NOTICE
```

```
#define     RPCR_EVICTION_MUST_UPGRADE
#define     RPCR_EVICTION_END_MEMBERSHIP
#define     RPCR_EVICTION_NO_PAYMENT
```

DBReleaseAccessToken
DBReleaseAccessToken(long AccessTokenId)

IN AccessTokenId

Processing:

- Update the Usage table with the appropriate information.
- Delete the LicenseUsage record.

Notes:

- We need a mechanism to release un-released access tokens. The way to do this would be to run a stored procedure at demand and at a predefined intervals to do this cleaning up.

EvictAccessToken
DBReleaseAccessToken(long AccessTokenId)

IN AccessTokenId

Processing:

- Evicts an access token.

Billing Component

AddUsageRecord. Called by the SLM server when it releases an access token.

Bool AddUsageRecord(String username, int subscriptionId, date starttime, long duration).

GetUsageRecordsForUser. Used by external billing system.

Couple[][] GetUsageRecordsForUser(String username)

GetUsageRecordsForGroup Used by external billing system.

Couple[][]GetUsageRecordsForGroup (String groupName)

Application Management Component

AddApplication

int AddApplication(String appname, String version, String description)

Inputs:
Appaname: Application name.
Appversion: Application version
Description. Application description.
Outputs:
-1 for failure to add the application.
>0 otherwise. Application ID.
Comments:
Returns an app id for a newly added application.
Errors:
APPLICATION EXISTS
NO DATABASE CONNECTION
UNKNOWN SQL ERROR

GetApplicationId

int GetApplicationId(String appname, int version)

Inputs:
Appaname: Application name.
Appversion: Application version
Outputs:
-1 for failure to find the application.
>0 otherwise.
Comments:
Returns an app id for a given application.
Errors:
NO DATABASE CONNECTION
UNKNOWN SQL ERROR

GetApplicationId

int GetApplicationId(int SubscriptionId)

Inputs:
SubscriptionId
Outputs:
0 for failure to find the application.

>0 otherwise.

Comments:

Returns an app id for a given application.

Errors:

NO DATABASE CONNECTION
UNKNOWN SQL ERROR

GetSubscribedApplicationIds

Int[]* GetSubscribedApplicationId(String username)

Inputs:

Username: username.

Outputs:

Array of application ids subscribed by a user.

Comments:

Errors:

USER NOT FOUND
NO DATABASE CONNECTION
UNKNOWN SQL ERROR

GetUnsubscribedApplicationIds

Int[]* GetUnsubscribedApplicationId(String username)

Inputs:

Username: username.

Outputs:

Array of application ids not subscribed by a user.

Comments:

Errors:

USER NOT FOUND
NO DATABASE CONNECTION
UNKNOWN SQL ERROR

Couple[] GetApplicationDetail(int appid)

Inputs:

Application Id.

Outputs:

Array of couple for the app id containing:
{appname, appversion, description} values.

Comments:

Errors:

```
USER NOT FOUND
NO DATABASE CONNECTION
UNKNOWN SQL ERROR
```

Component design

We will discuss some complex scenarios in this section.

Subscription

Single New User

1. Create the user. **CreateUser.**
   a. If a user already exists, return error message and go back to 1.
2. Create the account for the user. **CreateAccount**
   a. Get the contact information from the user.
   b. Prompt to get the billing information. The user may decide to not give the billing information at this point.

Corporate group admin creating an account.

1. Create the admin user. **CreateUser.**
2. Create the group. **CreateGroup**
3. Create the account information for the group. **CreateAccount.**
   c. Get the contact information from the user.
   d. Prompt to get the billing information.
4. Add users to the group. **AddUserToGroup.**
   a. This method will automatically create the user if they do not already exist in the system.
   b. The list of users is accessible to the Group Admin by querying:
      i. Our database **GetUserRecords** OR
      ii. Some external database. E.g. LDAP directory.

Single User subscribing to an application

1. Validate the user. **ValidateUser**

2. Prompt to get the billing information if the billing information is not already present.
3. Get the list of un-subscribed applications. GetUnsubscribedApplications.
   a. **GetUnsubscribedApplicationIds.**
   b. For each app id returned, get the application details. **GetApplicationDetail**
4. For each additional application user wants to subscribe, call **AddSubscription**

SLiM server checking out an access token to use an application

1. Call **DBAcquireAccessToken.**

Testing design

This document must have a discussion of how the component is to be tested. Some subsections could include:

Unit testing plans

Stress testing plans

Coverage testing plans

Cross-component testing plans

Upgrading/Supportability/Deployment design

This document must have a discussion of how the component addresses any specific issues related to upgrading, supporting and deployment of e-stream applications. Some examples include: error conditions detected and reported by this component, any special hooks this component will provide for monitoring, hints for troubleshooting problems, any special hooks for debugging this component.

Open Issues

This is a list of issues that need to be further investigated or revisited during implementation.

Omnishift C/C++ Coding Standard

[redacted]

Omnishift Confidential

The following proposal is based on the C++ coding standards document available at `http://www.possibility.com/Cpp/CppCodingStandard.html`. This document will concisely present the coding standards from the coding standard document. The reader should refer to the original document (linked above) for a detailed explanation of the standards. This document has the following sections:

NAMES: This section contains the naming schemes.
ERRORS AND ERROR CODES: This section contains the error formats and the error codes to be used by eStream 1.0 client and server components.
FORMATTING: Code layout and formatting guidelines.
COMMENTS: Guidelines for applying comments to the code.
LOOSE END: Loose ends.

Table of Contents:

NAMES

ID	RULE
Variables	• Use upper case as word separators, lowercase for the rest of the word. • No underbars ('_') • First letter of the variable could be upper/lower case. **Examples:** short Status; unsigned long timeOfDay;
Pointers	• Prepend the variable with p. **Examples:** string* pName; char** ppValue;
Class Names	• Use upper case letters as word separators, lower case for the rest of a word • First character in a name is upper case • For externally exposed components, use the first 3 words to denote the component. • No underbars ('_') **Examples:** • class ConfigurationManager • class Config
Library Class Names	• Prefix the classname with OT **Examples:** • class OTHttpListener • class OTServer
Class Method Names	Same rule as for class names except for interfaces where the rule is: • Prefix the interface with the component's name. • Method names may optionally start with a lower case letter. **Examples:** • ECMGetFileId • MonitorGetServerSet • monitorInitialize.
Class Attribute Names	• Attribute names should be prepended with the character 'm'. • After the 'm' use the same rules as for class names. • 'm' always precedes other name modifiers like 'p' for pointer. Examples: int mLen; char* mpName; string* mpValue;
Global Variables	• Global variables should be prepended with "g". Examples: int gFlag; Logger gLog;

	Logger* gpLog;
Global Constants	• Global constants should be all caps with '_' separators. Examples: const int A_GLOBAL_CONSTANT= 5;
#defines and Macros	• Put #defines and macros in all upper using '_' separators Examples: #define MAX(a,b) blah #define IS_ERR(err) blah
Function Names	• Use upper case letters as word separators, lower case for the rest of a word • First character in a name is upper case • No underbars ('_') **Examples:** • int SomeBloodyFunction()

Enum Names	• all upper using '_' separators
	Examples:
	enum PinStateType
	{
	PIN_OFF,
	PIN_ON
	}

File Names	• File should be all lower case • File name format should be <component>_<sub-component>.* **Examples:** monitor_heartbeat.cpp core_configmanager.c

ERRORS AND ERROR CODES

The following pound defines should be used for returning all successes and failures.

#define SUCCESSWITHINFO -1
#define SUCCESS 0
#define FAILURE >0 (The number representing an Error ID).

All error messages will be prepended with an error code. The format for error code will be as follows:

[ERRORID] [Severity] [Error Message]

where,

1. ERRORID are unique across the system.
2. Severity can be one of the following:
    a. 1-Low : A warning which can be ignored.
    b. 2-Medium: A warning which needs to be looked into.
    c. 3-High: Recoverable error in the component.
    d. 4-Critical: Irrecoverable error. Needs admin assistance.
3. The error message in itself should have the following format:
   [COMPONENT]:[ERROR MESSAGE]:[WORK AROUND]

Error Ids distribution for client and server are as follows:

0-1000 Server Internal Error Codes.
1001 – 8000 Server Error Codes.
8001 – 9000 Client Internal Error Codes.
9001 –16000 Client Error Codes.

FORMATTING

The following formatting policies should be followed by all code.

Braces Policy.

```
if ( 0 == a)
{
...
}
else
{
```

```
...
}
```

Indentation/Tabs/Space Policy

Use the standard Visual C++ settings which are(using Tools->Options->Tabs menu):

Indent Size: 4
Auto Indent: Smart
100 previous lines used for context.

White spaces should be spaces and NOT tabs.
VC++: Select "Insert Spaces" option. (This is NOT the default).
Emacs: Refer http://www.delorie.com/gnu/docs/emacs/emacs_205.html

Line Size etc.

1. Line size should not exceed 78 characters.
2. There should be one statement per line. The following piece of code violates this principle.

```
if (a>b) a++;
```

Method/Functions Formats.

1. Methods should preferably be less 50 lines of code.
2. Methods should not have more than 4 levels of nesting.
3. Methods should preferably be re-entrant. Non-reentrant methods should be clearly marked as such.
4. Each method/function should be preceded with a comment describing the method:

```
/*****************************************************************
FUNCTION:
INPUTS:
OUTPUTS:
DESCRIPTION:
ERRORS:
*****************************************************************/
```

COMMENTS

Every file should start with the following Copyright disclaimer:

```
/* ================================================================
 * The Omnishift Software License, Version 1.0
 *
 * [redacted]mnishift Technology.  All rights
 * reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are not permitted
 *
 */
```

1.
2. Every decision should have comments. The following keyword are associated with decisions:
    a. if, else
    b. while, continue
    c. switch, case, default, break
    d. goto
    e. return
3. Every class should have comment header with the following format:

```
/*****************************************************************
CLASS NAME:
DESCRIPTION:
FRIEND CLASSES:
INCLUDES:
LIBRARIES:
*****************************************************************/
```

4. Every function/method should have a header. (described above).
5. Every file should have a header describing the contents of the file.
6. Every directory should have a README describing the contents of the directory.
7. Make GOTCHAS explicit. Use the following format for gotchas.

- **:TODO: topic** *<Author>:<Date>*
  Means there's more to do here, don't forget.
- **:BUG: [bugid] topic** *<Author>:<Date>*
  means there's a Known bug here, explain it and optionally give a bug ID.
- **:HACK:** *<Author>:<Date>*
  When you've done something ugly say so and explain how you would do it differently next time if you had more time.
- **:TRICKY:** *<Author>:<Date>*
  Tells somebody that the following code is very tricky so don't go changing it without thinking.

- **:WARNING:** *<Author>:<Date>*
  Beware of something.
- **:COMPILER:** *<Author>:<Date>*
  Sometimes you need to work around a compiler problem. Document it. The problem may go away eventually.
- **:ATTRIBUTE: value** *<Author>:<Date>*
  The general form of an attribute embedded in a comment. You can make up your own attributes and they'll be extracted.

where an example of the Author:Date format would be: Bhaven Avalani:[redacted]

LOOSE ENDS

The following section notes some loose ends which do not fall in any of the categories above:

1. Always **initialize all variables** every time.
2. Use **header file guards** against multiple inclusions of the header file. The guards would look like:

   ```
   #ifndef ClassName_h
   #define ClassName_h
   ....
   #endif // ClassName_h
   ```
3. Object constructors should just initialize data. (They cannot return errors). Explicit Initialize() calls should be made to do any involved work.
4. Use **continue** and **goto** sparingly.
5. Be **"const" correct**. Use "const" wherever and whenever applicable.
6. All classes must have a **Default Constructor** and a **Copy Constructor**
7. Set the compilation flag "Warnings as error" in Project -> Settings -> C/C++. This will show all warnings as errors.
8. Use std:: namespace for all STL classes.

//develop/eng/docs/readme.txt

This directory is for keeping all engineering related documents,
but not product specific documents. Product specific documents are
to be created with the product <docs> directory.

Software Development Process

Version 0.2
Ricky Benitez
[redacted]

We are what we repeatedly do,
Excellence, then, is not an act, but a habit.

ARISTOTLE

This document describes the software development process followed at OTI for all software products. The purpose of this process is to ensure that software products are developed in an effective, predictable, repeatable and continuously improvable manner. The process has inputs, methods, outputs and metrics to determine its effectiveness.

Ownership

The overall responsibility for the development, refinement, effectiveness and adherence to the software development process belongs to the VP of Engineering. Responsibility for the various steps in the process will be assigned to appropriate individuals depending on the scope of the product and the makeup of the development group.

Inputs

The inputs to the software development process are:

1. Marketing Requirements Document (MRD)
2. Technology White Papers and Prototypes

Marketing Requirements Document

The VP of Marketing is responsible for the development of this document. The VP of Engineering is responsible for extracting the portions of the MRD that will be implemented in software and capturing these as the High Level Requirements document.

Technology White Papers

The Chief Technology Officer is responsible for providing any technology white papers and prototypes that provide proof of concept and suitable technological direction needed to convert the marketing requirements into a software product.

Methods

The primary methods used during the development process are described in detail later in the document, but primarily consist of:

1. Transforming the MRD to the High Level Requirements document (HLR)
2. Transforming the HLR to the Product Datasheet (PDS)

3. Transforming the HLR to the High Level Design document (HLD)
4. Transforming the HLR to the Product Test Plan (PTP)
5. Transforming the HLD to the Low Level Design documents (LLDs)
6. Transforming the HLD to an End-to-end Test Infrastructure (ETI)
7. Transforming the LLDs to an Integration Test Infrastructure (ITI)
8. Transforming the LLDs to Product Source Code (PSC)
9. Integration
10. Final Validation
11. Performing a Post Mortem

Outputs

The outputs of the software development process are:

1. Updated MRD
2. Product Datasheet (PDS)
3. Digital components that meet the requirements listed in the HLR
4. A set of mutually consistent and appropriately labeled design documents, source code, build environment, build infrastructure, test plans and test harnesses in a revision control system
5. An issue database which contains a description of all past and outstanding issues relating to the requirements, design and implementation of the software components of the product
6. A post mortem report indicating what was learned during the development process that can be used to improve the process in the future.

Metrics

Metrics are needed to track the progress of the development process and to constantly improve and fine-tune it. The primary mechanisms employed to measure are:

1. Completion of the methods
2. The product issue database
3. The process issue database

These will be discussed in detail later in this document.

Overview of the Process

The software development process is iterative. While it may be described in a linear fashion, it must be understood that events will often dictate that previously completed portions of the process need alterations and that these alterations may propagate forward or backward along the various stages of the process. The ultimate objective is to end up with a set of outputs that are consistent relative to each other and relative to the inputs.

Supporting Documents

The following documents exist in conjunction with this one to support and fully define the overall software development process:

1. Coding Guidelines
2. Configuration and Release Management Guidelines

Description of the Methods

MRD to HLR

The input of this method is a marketing requirements document and its output is a high-level requirements document. This method is performed once on a static MRD and then iterates incrementally as events warrant. Every change to the MRD after the initial HLR is produced generates an issue. The HLR must be kept in synch with the MRD and vice-versa. Once the initial HLR is produced changes are made only as a step towards resolving an issue.

Description

The HLR is a set of precise imperatives that collectively define the scope and behavior of the software product. Each precise imperative is known as a requirement. Requirements are grouped into the following categories:

1. Functionality
2. Localization
3. Usability
4. Reliability
5. Performance
6. Scalability
7. Security
8. Portability
9. Maintainability

Every product will have at least one requirement within each of these categories. Each requirement consists of the following:

1. Unique number – used to identify the requirement in other documents
2. Description – a concise description of the requirement
3. Importance – the level of importance to the final product between 1 and10. An importance of 1 indicates that the failure to fully or partially meet that requirement will have very minimal impact on the success of the product. An importance of 10 indicates that the product must either be able to meet the specified requirement or should not be produced at all

Use cases are also included in an HLR to describe the various scenarios that the product is expected to handle from the perspective of its various users. Use cases are presented with the following information:

1. Summary – a description of the use or activity
2. Actors – a listing of those who have a role in the activity
3. Inputs – the initial state of the product and additional data that the actors need to
4. Processing – the sequence of steps taken by the actors
5. Output – the final state of the system and any data that the system produces

Template

An HLR template is located in the OTI share under \\fserv1\oti\general\templates.

Review

Upon completion of the HLR, a requirements review will be held. The requirements review process is as follows:

1. No less than four primary reviewers are selected, one should be from the engineering team (preferably the architect of the system), one should be from product marketing, one should be from the release group and one should be from the deployment or PSO organization.
2. The primary reviewers are given a copy of the MDR and the HLR document sufficiently prior to the review meeting to review the HLR.
3. The primary reviewers ensure to the best of their ability that:
    - The requirements are precise
    - There are appropriate requirements for every category (or a good understanding as to why no requirement is needed for a particular category)
    - The requirements and their importance level will, if faithfully transformed into a software product and incorporated with appropriate marketing, deployment and support, result in a successful customer solution
4. At the start of the review, a scribe (not a primary reviewer) is selected. The scribe notes down who is in attendance and ensures that all primary reviewers are present and have reviewed the requirements.
5. If the requirements have not been fully reviewed by the primary reviewers, the review must be rescheduled.
6. Primary reviewers bring up their issues and a decision is made as to whether the item is or is not a real issue. The scribe logs all real issues.
7. When the primary reviewers have presented all their issues, other attendees are invited to bring up issues. Real issues are logged.
8. The scribe then polls the primary reviewers to determine which of the following courses of action will be taken:
    - HLR is completed after all issues are satisfactorily resolved
    - HLR must be reviewed again after all issues are satisfactorily resolved
    - HLR must be abandoned and a new requirements effort begun
9. After the meeting, the scribe is responsible for filing and submitting the review report. If the HLR is not to be abandoned, the scribe is also responsible for entering all issues into the issue tracking system.

Completion

The MDR to HLR method is considered complete only after:

- a review of the HLR whose primary reviewers decide should be considered completed has taken place
- a review report has been submitted for every review of the HLR
- all HLR issues have been submitted and satisfactorily resolved
- the HLR has been appropriately added to the revision control system

Issue Resolution

HLR issues that result in the removal, change or inclusion of one or more requirements can be considered resolved after the HLR and any other document that are affected and must be kept in synch with the HLR are appropriately updated and each change has been reviewed and approved by one reviewer from the engineering team, one from the marketing team, one from the product marketing group, one from the release group and one from the deployment or PSO group. If the HLR is significantly changed as a result of addressing an issue, a full review should be considered.

HLR to PDS

The input to this method is a high-level requirements document and the output is a product datasheet. This method is performed once on the HLR and then iterates incrementally as events warrant. The PDS must be kept in synch with the HLR and vice-versa. Once the initial PDS is produced changes are primarily made only as a step towards resolving an issue. The speculative nature of the PDS gives its owner more leeway during review and editing than is accorded to the owners of most other product documents and deliverables.

Description

The PDS is a document consisting of various sections that describe the overall software product to potential stakeholders. The PDS contains a section for each of the following:

1. Is/Is Not – this section describes the product in terms of what it is and, to dispel potential misconceptions about what it might be, what it is not
2. Flexibility Matrix – this section indicates the relative importance of time, features and cost
3. Requirements – this section is a copy of the HLR, potentially tailored to a less technical audience
4. Schedule – this section lists the major milestones of the product
5. Dependencies – this section lists the major external dependencies of the product
6. Resources – this section lists the resources needed by engineering to complete the product as follows:
    a. Engineers and managers needed per month
    b. Machine resources needed
    c. Software resources (licenses, etc.) needed
7. Quality Plan – this section lists the steps that will be taken towards reaching the quality goals of the product:
    a. What will be minimally prototyped prior to high-level design
    b. Any exceptions or additions to the software development process defined
    c. Coverage, number of white-box tests and other criteria required to meet completion on PSC and final validation
    d. Alpha, beta and other pre-production releases
8. Risks – this section lists the anticipated risks and appropriate contingency plans to address those risks.

Template

A PDS template is located in the OTI share under \\fserv1\oti\general\templates.

Review

Upon completion of the PDS, it is posted for the managers, engineers and stakeholders of the product to review.

Completion

The HLR to PDS method is considered complete only after:

- The PDS has been completely filled to the best understanding of its owner
- The PDS has been posted for review

Issue Resolution

PDS issues that result in changes to the PDS are considered resolved after the PDS and any other document affected that must be kept in synch with the PDS are appropriately updated and reviewed.

HLR to HLD

The input of this method is a high-level requirements document and the output is a high-level design document. This method is performed once on the HLR and then iterates incrementally as events warrant. The HLD must be kept in synch with the HLR and vice-versa. Once the initial HLD is produced changes are made only as a step towards resolving an issue.

Description

The HLD is a document consisting of various sections that describe the overall software product in high-level form. The HLD contains a section for each of the following:

1. Definition of terms – this section defines each technical term that may lead to confusion in the understanding of the design if left undefined
2. Block diagram – a pictorial description of the system, to aid in the identification and understanding of the components and APIs described in the design
3. High-level description of each component – a description of each major component of the system which clearly describes the role that each component plays in the overall system and serves as the launching point of the component's low level design
4. High-level description of each API – a catalog and simple description of each major API in the system, which a particular emphasis on the system's external [illegible]
5. High-level test strategy – a description of the approach to be taken to validating the correctness of the system end-to-end

Template

An HLD template is located in the OTI share under \\fserv1\oti\general\templates.

Review

Upon completion of the HLD, a low-level design review will be held. The HLD review process is as follows:

1. No less than two primary reviewers are selected.
2. The primary reviewers are given a copy of the HLR and the final HLD document sufficiently prior to the review meeting to review the HLD.
3. The primary reviewers ensure to the best of their ability that:
    - The design meets its requirements
    - All sections of the design are complete
    - The design is as simple, robust, testable, scalable and well thought through as time requirements permits
    - Competent designers could perform a low-level design of each component given the HLR and HLD without needing to consult the author of the HLD
4. At the start of the review, a scribe (not a primary reviewer) is selected. The scribe notes down who is in attendance and ensures that all primary reviewers are present and have reviewed the design.
5. If the design has not been fully reviewed by the primary reviewers, the review must be rescheduled.
6. Primary reviewers bring up their issues and a decision is made as to whether the item is or is not a real issue. The scribe logs all real issues.
7. When the primary reviewers have presented all their issues, other attendees are invited to bring up issues. Real issues are logged.
8. The scribe then polls the primary reviewers to determine which of the following courses of action will be taken:
    - Design is completed after all issues are satisfactorily resolved
    - Design must be reviewed again after all issues are satisfactorily resolved
    - Design must be abandoned and a new design effort begun
9. After the meeting, the scribe is responsible for filing and submitting the review report. If the design is not to be abandoned, the scribe is also responsible for entering all issues into the issue tracking system.

Completion

The HLR to HLD method is complete only after:

- a review of the HLD whose primary reviewers decide should be considered completed after all issues are resolved has transpired
- a review report has been submitted for every review of the HLD
- all HLD issues have been submitted and satisfactorily resolved
- any changes required to the HLR to bring both HLR and HLD documents in synch have been submitted as issues
- the HLD has been appropriately added to the revision control system

Issue Resolution

HLD issues that require changes to the HLD can be considered resolved after the HLD document and any other documents that are affected and must be kept in synch with the HLD are appropriately updated and each change has been reviewed and approved by at

least one reviewer. If the HLD is significantly changed as a result of addressing an issue, a full review should be considered.

HLR to PTP

The input to this method is the high-level requirements document and the output is a product test plan. This method is performed once on a static HLR and iterates incrementally as events warrant. The PTP must be kept in synch with the HLR and vide-versa. Once an initial PTP is produced changes are made only as a step towards resolving an issue.

Description

The PTP is a document describing the various black-box and stress tests that will be applied to the product to ensure that it meets its stated requirements. The PTP should completely cover every requirement in the HLR to ensure that the delivered product meets its stated goals.

Template

TBD

Review

Upon completion of the PTP, a test plan review will be held. The PTP review process is as follows:

1. No less than two primary reviewers are selected.
2. The primary reviewers are given a copy of the final PTP document sufficiently prior to the review meeting to review the PTP.
3. The primary reviewers ensure to the best of their ability that:
    - There are appropriate tests defined to cover all the requirements
    - The test are as simple, robust, repeatable, scalable and well thought through as time requirements permits
    - A competent SQA engineer could conduct the test using the PTP without needing to consult its author
    - The PTP meets the quality criteria stated in the quality plan section of the PDS
4. At the start of the review, a scribe (not a primary reviewer) is selected. The scribe notes down who is in attendance and ensures that all primary reviewers are present and have reviewed the test plan.
5. If the test plan has not been fully reviewed by the primary reviewers, the review must be rescheduled.
6. [illegible] item is or is not a real issue. The scribe logs all real issues.
7. When the primary reviewers have presented all their issues, other attendees are invited to bring up issues. Real issues are logged.
8. The scribe then polls the primary reviewers to determine which of the following courses of action will be taken:
    - Plan is completed after all issues are satisfactorily resolved

- Plan must be reviewed again after all issues are satisfactorily resolved
- Plan must be abandoned and a new planning effort begun

9. After the meeting, the scribe is responsible for filing and submitting the review report. If the plan is not to be abandoned, the scribe is also responsible for entering all issues into the issue tracking system.

Completion

The HLR to PTP method is complete only after:

- a review of the PTP whose primary reviewers decide should be considered completed after all issues are resolved has transpired
- a review report has been submitted for every review of the PTP
- all PTP issues have been submitted and satisfactorily resolved
- any changes required to the HLR to bring this documents in synch with the PTP have been submitted as issues
- the PTP has been appropriately added to the revision control system

Issue Resolution

PTP issues that require changes to the PTP can be considered resolved after the PTP document and any other documents that are affected and must be kept in synch with the PTP are appropriately updated and each change has been reviewed and approved by at least one reviewer. If the PTP is significantly changed as a result of addressing an issue, a full review should be considered.

HLD to LLDs

The input to this method is a high-level design document and the output is one low-level design document for each component described in the HLD. This method is performed once on the HLD and then iterates incrementally as events warrant. The LLD must be kept in synch with the HLD and vice-versa. Once the initial LLD is produced changes are made only as a step towards resolving an issue.

Description

The LLD is a document consisting of various sections to describe the low-level design of a component. The LLD contains a section for each of the following:

1. Functionality – this section describes the functionality that this component provides
2. Data type definitions – a list of the data types defined or used by this component
3. Data structures – a description of the data structures defined and used by this component
4. Interface definitions – a description of the external and internal interfaces supported by this component — for user interfaces, a mock-up or prototype of the interface must be provided (separate from the LLD) and appropriate screen shots of this prototype should be included
5. Component design description – a description of the component sufficient to allow a competent programmer to implement the component without needing to consult the original author

6. Testing – a description of the test harnesses used to unit test, stress test and/or coverage test this component sufficient to allow a competent programmer to implement these harnesses without consulting the author
7. Supportability – a description of the supportability features of the component such that an integration engineer can make full use of the supportability features built into the component without needing to consult the author or developer

Template

An LLD template is located in the OTI share under \\fserv1\oti\general\templates.

Review

Upon completion of the LLD, a low-level design review will be held. The LLD review process is as follows:

1. No less than two primary reviewers are selected.
2. The primary reviewers are given a copy of the final LLD document sufficiently prior to the review meeting to review the LLD.
3. The primary reviewers ensure to the best of their ability that:
    - The design meets its requirements
    - All sections of the design are complete
    - The design is as simple, robust, testable, scalable and well thought through as time requirements permits
    - A competent software engineer could implement the component using the LLD without needing to consult its author
4. At the start of the review, a scribe (not a primary reviewer) is selected. The scribe notes down who is in attendance and ensures that all primary reviewers are present and have reviewed the design.
5. If the design has not been fully reviewed by the primary reviewers, the review must be rescheduled.
6. Primary reviewers bring up their issues and a decision is made as to whether the item is or is not a real issue. The scribe logs all real issues.
7. When the primary reviewers have presented all their issues, other attendees are invited to bring up issues. Real issues are logged.
8. The scribe then polls the primary reviewers to determine which of the following courses of action will be taken:
    - Design is completed after all issues are satisfactorily resolved
    - Design must be reviewed again after all issues are satisfactorily resolved
    - Design must be abandoned and a new design effort begun
9. After the meeting, the scribe is responsible for filing and submitting the review entering all issues into the issue tracking system.

Completion

The HLD to LLD method is complete only after:

- a review of the LLD whose primary reviewers decide should be considered completed after all issues are resolved has transpired

- a review report has been submitted for every review of the LLD
- all LLD issues have been submitted and satisfactorily resolved
- any changes required to the HLD to bring this document in synch with the LLD have been submitted as issues
- the LLD has been appropriately added to the revision control system

Issue Resolution

LLD issues that require changes to the LLD can be considered resolved after the LLD document and any other documents that are affected and must be kept in synch with the LLD are appropriately updated and each change has been reviewed and approved by at least one reviewer. If the LLD is significantly changed as a result of addressing an issue, a full review should be considered.

HLD to ETI

The input to this method is a high-level design document with a high-level test strategy description and the output is an end-to-end test infrastructure. This method is performed once on the HLD and then iterates incrementally as events warrant. The ETI infrastructure must be kept in synch with the high-level test strategy in the HLD and vice-versa. Once an initial ETI is produced changes are made only as a step towards resolving an issue.

Description

The ETI is a set of scripts, source files, include files, make files and documents that collectively are used to automate the process of performing a rigorous end-to-end test of the integrated PSC. The purpose of the ETI is to provide an automated method for ensuring that the overall integrated PSC operates on a regular (nightly) basis.

Template

TBD

Review

Upon completion of the ETI, a code review will be held. The code review process is as follows:

1. No less than two primary reviewers are selected.
2. The primary reviewers are given a copy of the ETI and related code (or pointed to their location in the configuration management system) sufficiently prior to the review meeting to review the code.
3. The primary reviewers ensure to the best of their ability that:
    - The code is an efficient, reasonable and complete implementation of the high-level test strategy. Efficiency means that the end-to-end test does not require manual steps and can complete in no more than a couple of hours.
    - A competent SQA engineer could maintain the ETI without needing to consult its author (assuming that the HLD is also available)

4. At the start of the review, a scribe (not a primary reviewer) is selected. The scribe notes down who is in attendance and ensures that all primary reviewers are present and have reviewed the implementation.
5. If the implementation has not been fully reviewed by the primary reviewers, the review must be rescheduled.
6. Primary reviewers bring up their issues and a decision is made as to whether the item is or is not a real issue. The scribe logs all real issues.
7. When the primary reviewers have presented all their issues, other attendees are invited to bring up issues. Real issues are logged.
8. The scribe then polls the primary reviewers to determine which of the following courses of action will be taken:
    - Implementation is completed after all issues are satisfactorily resolved
    - Implementation must be reviewed again after all issues are satisfactorily resolved
    - Implementation must be abandoned and a new implementation effort begun

After the meeting, the scribe is responsible for filing and submitting the review report. If the implementation is not to be abandoned, the scribe is also responsible for entering all issues into the issue tracking system.

Completion

The HLD to ETI method is complete only after:

- a review of the ETI whose primary reviewers decide should be considered completed after all issues are resolved has transpired
- a review report has been submitted for every review of the ETI
- all ETI issues have been submitted and satisfactorily resolved
- any changes required to the HLD test strategy to bring this documents in synch with the ETI have been submitted as issues
- the ETI has been appropriately added to the revision control system as specified in the configuration and release management guidelines
- the ETI has been set up to run automatically on each successful and sanity-tested build of the integrated PSC and the results of the test are being mailed to all interested parties

Issue Resolution

ETI issues that require changes to the test infrastructure must be appropriately reviewed before they are closed. The review process depends on the nature of the changes made to the ETI. If only one to twenty-five source statements are affected, then a desk check [illegible] member needs to add their review comments to the issue prior to closing it. If more than twenty-five statements are added or modified, then the standard review process for ETI must be followed.

LLD to ITI

The input of this method is the set of low-level design documents with component test strategies and the output is an integration test suite definition and infrastructure. This method is performed once after all or after a substantial portion of the LLDs have been produced and then iterates incrementally as events warrant. The ITI must be kept in synch with the LLDs and vice-versa. Once the initial ITI is produced changes are made only as a step towards resolving an issue.

Description

The ITI is a set of scripts, source files, include files, make files, test definition files and documents that collectively are used to automate the process of performing a rigorous set of regression, black-box, stress, coverage, performance and sanity tests on the PSC. The purpose of the ITI is to provide an automated method for ensuring that the individual product components and the overall integrated PSC is robust on a regular (nightly) basis.

Template

TBD

Review

Upon completion of the ITI, a test suite and code review will be held. The code review process is as follows:

1. No less than two primary reviewers are selected.
2. The primary reviewers are given a copy of the test suite and related code (or pointed to their location in the configuration management system) sufficiently prior to the review meeting to review the suite and the code.
3. The primary reviewers ensure to the best of their ability that:
    - The test suite will efficiently provide a high-level of confidence that every component of the PSC is of high quality. Efficiency means that the entire ITI runs in no more than a few hours. This entails examining each test suite and ensuring that tests are not overly redundant or spend inordinate amounts of time running while providing only marginal improvements in the confidence level of the product's quality.
    - The code is reasonable and complete automation of the integration test suite
    - A competent SQA engineer could maintain the ITI without needing to consult its author (assuming that the LLDs are also available)
4. At the start of the review, a scribe (not a primary reviewer) is selected. The scribe notes down who is in attendance and ensures that all primary reviewers are present and have reviewed the implementation.
5. If the implementation has not been fully reviewed by the primary reviewers, the review must be rescheduled.
6. Primary reviewers bring up their issues and a decision is made as to whether the item is or is not a real issue. The scribe logs all real issues.
7. When the primary reviewers have presented all their issues, other attendees are invited to bring up issues. Real issues are logged.

8. The scribe then polls the primary reviewers to determine which of the following courses of action will be taken:
    - Test suite and implementation is completed after all issues are satisfactorily resolved
    - Test suite and implementation must be reviewed again after all issues are satisfactorily resolved
    - Test suite and implementation must be abandoned and a new implementation effort begun

After the meeting, the scribe is responsible for filing and submitting the review report. If the test suite and implementation is not to be abandoned, the scribe is also responsible for entering all issues into the issue tracking system.

Completion

The HLD to ITI method is complete only after:

- a review of the ITI whose primary reviewers decide should be considered completed after all issues are resolved has transpired
- a review report has been submitted for every review of the ITI
- all ITI issues have been submitted and satisfactorily resolved
- any changes required to the LLD test strategies to bring this documents in synch with the ITI have been submitted as issues
- the ITI has been appropriately added to the revision control system as specified in the configuration and release management guidelines
- the ITI has been set up to run automatically after each successful build of the integrated PSC and the results of the test are being mailed to all interested parties

Issue Resolution

ITI issues that require changes to the test suite or the source code must be appropriately reviewed before they are closed. The review process depends on the nature of the changes made to the ITI. If changes are one to several additional tests added to an existing test suite, then a desk check of each added test by another member of the technical staff is all that is required. If a new test suite or more than several new test cases are being added, then the standard review process for the ITI must be followed. If changes are to source code and only one to twenty-five source statements are affected, then a desk check review performed by another member of the technical staff is all that is required. That member needs to add their review comments to the issue prior to closing it. If more than twenty-five statements are added or modified, then the standard review process for ITI must be followed.

The input to this method is a low-level design document and the output is product source code. This method is performed once for each completed LLD and then iterates incrementally as events warrant. The LLD must be kept in synch with the PSC and vice-versa. Once the initial PSC is produced changes are made only as a step towards resolving an issue.

Description

The PSC is a collection of the source files, include files, make files and a collection of development tools and an environment that will be used to produce the binary bits that will be ultimately delivered to a customer. Additional supporting code including, but not limited to code generators, component tests, white box tests, regression tests and test drivers are produced while applying this method. PSC must adhere to the coding guidelines and the completion criteria for this method. PSC and all other supporting code must adhere to the configuration and release management guidelines.

Template

See the coding guidelines and configuration and release management guidelines for template information.

Review

Upon completion of the PSC, a code review will be held. The code review process is as follows:

1. No less than two primary reviewers are selected.
2. The primary reviewers are given a copy of the PSC and related code (or pointed to their location in the configuration management system) sufficiently prior to the review meeting to review the code.
3. The primary reviewers ensure to the best of their ability that:
    - The code is a reasonable and complete implementation of the LLD
    - The associated white-box and component tests specified in the LLD have been implemented and have passed successfully
    - The PSC adheres to the coding guidelines
    - A competent software engineer could maintain the PSC without needing to consult its author (assuming that the LLD and HLD are also available)
4. At the start of the review, a scribe (not a primary reviewer) is selected. The scribe notes down who is in attendance and ensures that all primary reviewers are present and have reviewed the implementation.
5. If the implementation has not been fully reviewed by the primary reviewers, the review must be rescheduled.
6. Primary reviewers bring up their issues and a decision is made as to whether the item is or is not a real issue. The scribe logs all real issues.
7. When the primary reviewers have presented all their issues, other attendees are invited to bring up issues. Real issues are logged.
8. The scribe then polls the primary reviewers to determine which of the following courses of action will be taken:
    - Implementation is completed after all issues are satisfactorily resolved
    - Implementation must be reviewed again after all issues are satisfactorily resolved
    - Implementation must be abandoned and a new implementation effort begun

After the meeting, the scribe is responsible for filing and submitting the review report. If the implementation is not to be abandoned, the scribe is also responsible for entering all issues into the issue tracking system.

Completion

The LLD to PSC method is complete only after:

- a review of the PSC and associated code whose primary reviewers decide should be considered completed after all issues are resolved has transpired
- a review report has been submitted for every review of the PSC
- all PSC and associated code issues have been submitted and satisfactorily resolved
- any changes required to the LLD to bring this documents in synch with the PSC have been submitted as issues
- the PSC and associated code has been appropriately added to the revision control system as specified in the configuration and release management guidelines
- the PSC meets the coverage and component testing criteria specified in the quality plan section of the PDS

Issue Resolution

PSC issues that require changes to the source code must be appropriately reviewed before they are closed. The review process depends on the nature of the changes made to the PSC. If only one to twenty-five source statements are affected, then a desk check review performed by another member of the technical staff is all that is required. That member needs to add their review comments to the issue prior to closing it. If more than twenty-five statements are added or modified, then the standard review process for PSC must be followed.

Integration

The input to this method is the separate components product source codes, the integration test infrastructure and the end-to-end test infrastructure and the output is the fully integrated product source code. This method is performed iteratively starting as soon as there is more than one interacting component PSC available and as long as there are changes made to any of the component PSC. The method must take place as frequently as is practically possible while the PSC is changing. The desired goal is once per day.

Description

Integration is the process of bringing all of the PSC components together to produce the entire product and validating them against a suite of regression, black-box, stress, performance, entry tests and the ETI. A successful integration is a potential candidate for final validation. In order to provide developers with timely feedback concerning any integration and ETI failures that are introduced into the PSC, the integration process should be automated and performed as regularly as it is practical (every night, assuming that changes have been made to the PSC since the last successful integration).

Template

TBD

Review

Integration is by definition reviewed every time it is attempted (about once a day).

Completion

Integration is complete and a candidate for final validate is produced when:

- The entire system has been successfully built from the completed PSC of every component using only the documented build environment
- The built system successfully passes all regression, black-box, stress, performance, sanity and ETI test suites designated for integration
- All high and medium priority requirements, design and implementation issues have been resolved

Issue Resolution

There are rarely any issues against the integration phase. Most issues are ITI and ETI issues and are covered under those particular items. Issues might come up, however, in which integration is not taking place with the level of efficiency or producing the expected amount of confidence. In such cases, the issue may be categorized as an integration issue as a placeholder for determining whether it should be filed against ITI or ETI.

Final validation

The input to this method is a successful integration that produces a valid final validation candidate and the output is a fully validated release candidate. This method is applied as often as required, although it tends to be planned to coincide with code freeze and change control to ensure a quick and timely resolution of just those issues that are preventing the successful final validation of the product.

Description

Final validation is the process of taking a valid final validation candidate and applying the entire PTP against it. If every test defined in the PTP is successfully passed, then the candidate is declared to be a valid release candidate. Once this occurs, the build environment and all of the source files, include file, make files, test definition files and scripts that were used to produce the release candidate are appropriately labeled as stated in the configuration and release management guidelines.

Template

TBD

Review

Final validation is by definition reviewed every time it is attempted.

Completion

The integration stage is complete and a candidate for release is produced when:

- The entire PTP is successfully applied against a valid final validation candidate.
- The build environment and all of the source files, include file, make files, test definition files and scripts that were used to produce the release candidate are appropriately labeled as stated in the configuration and release management guidelines.
- No requirements, design, implementation or other types of issues logged against the current release with a priority greater than that stated as a requirement for release in the PDS remain unresolved.

Issue Resolution

There are rarely any issues against the final validation phase. Most issues are PTP issues and are covered under that category.

Post Mortem

The input to this method is a complete cycle through the software development process and the opinions of the participants. The post mortem method is applied once for each completed product release. The output is a post mortem report and a set of issues submitted against the software development process and any of the other supporting processes (such as the configuration and release management process).

Description

A post mortem captures the learning that took place over the software development lifecycle and should produce a number of issues against the software development process and other related processes that are to be resolved. If this is done soon after the completion of the development and release cycle and issues are addressed, then there is opportunity for continual refinement and improvement of the software development process and of the organization's development capability. The method requires that each participant fill out a feedback form and that these forms be reviewed by a set of individual who produce a post mortem report and file the issues that the reviewers determined were relevant against each appropriate process.

Template

TBD

Review

Upon completion of the post mortem feedback forms, a post mortem review will be held. The review process is as follows:

1. No less than two primary reviewers are selected.
2. All individuals who were asked to give feedback are invited to the review.
3. The primary reviewers are given a copy of the feedback forms sufficiently prior to the review meeting to review the code.
4. The primary reviewers ensure to the best of their ability that:

- All feedback comments are considered and appropriate root causes of issues are identified
- The issue tracking system is consulted to collaborate or disprove potential issues when appropriate

5. At the start of the review, a scribe (not a primary reviewer) is selected. The scribe notes down who is in attendance and ensures that all primary reviewers are present and have reviewed the implementation.
6. If the feedback has not been fully reviewed by the primary reviewers, the review must be rescheduled.
7. Primary reviewers bring up their issues and a decision is made as to whether the item is or is not a real issue. The scribe logs all real issues.
8. When the primary reviewers have presented all their issues, other attendees are invited to bring up issues. Real issues are logged.

After the meeting, the scribe is responsible for filing and submitting the post mortem report and entering all issues into the issue tracking system.

Completion

The post mortem is not complete until:

- Post mortem feedback has been received from all available participants
- The post mortem review is held
- All issues have been submitted against the appropriate process
- A post mortem report has been submitted

Issue Resolution

All issue filed against the software development process after the post mortem must be resolved in a timely manner. All changes must be reviewed and approved by the VP of Engineering.

Description of Metrics

TBD

THIS PAGE BLANK (USPTO)

Estream 1.0 Planning Document

[redacted]

Low-Level Design Status/Plan

Sub Components	Owner	LLD Design Doc completed	LLD review Compited	Estimates for Impl	Impl and Unit Test Completed
Content					
Install Monitor	Sanjay	Done	Done	3 wk	
Builder GUI	Sanjay	Done	Done	1 wk	
FSRFD (Drivers)	Sanjay	Done	Done	2 wk	
AppInstallBlk structure	David	Done	Not needed		
Profiler	David	Done	Done	2 wk	
File Access Monitor	David	Done	Done	1 wk	
Packager	David	Done	Done	1 wk	
eStream distribution	Bob	[redacted]	Status TBD	TBD	
Server Group					
Web Server	Bhaven	Done	Done	8 wk	
Monitor	Mike	Done	Done	4 wk	
SLIM Server	Amit	Done	Done	2 wk	
App Server	Sameer	Done	Done	4 wk	
Admin UI	Bhaven	TBD	TBD	TBD	
End User UI	Bhaven	TBD	TBD	TBD	
Common Server Components	Mike	Done	Done	3 wk	
Messaging	Sameer	Done	Done	3 wk	
Threads Package	Sameer	No Document		1 wk	
Security Design	Igor/Amit	Not Done	Not Done	TBD	
Client Group					
Cache Prefetching	Anne	Done	Done	1 wk	[redacted]
LSM + Plug In	Anne	Done	Done	1 wk	
Client UI	Anne	Done	Done	1 wk	
Client Installer	Anne	Done	Done	1 wk	
Start Client	Anne	Done	Done	1 wk	
Application Install Mgr	Nick	Done	Done	TBD	
Piracy	Nick	Done	Done	TBD	
File Spoofer	Curt	Done	Done	1 wk	
eStream File System	Curt	Done	Done	8 wk	
NoCluster Driver	Curt	Status TBD	Starus TBD	2 days	
eStream Cache Manager	Dan	Done	Done	8 wk	
Client Network Interface	Dan	Done	Done	2 wk	

Implementation Plan

Milestones

[redacted]

ECM (RAM disk cache) and EFSD executes a local "himom" executable
Photoshop is installed locally and successfully executed from estream sets and appinstallblk produced by builder
App Server and EMS integrated to copy "himom" executable using a dummy client
App Server, EMS and CNI integrated to copy "himom" executable from "himom" estream sets
office is installed locally and successfully executed from estream sets and appinstallblk produced by builder
App Server, EMS, ENI, ECM and EFSD integrated to run "himom" from estream sets on server
Following applications built and tested with local installation

- ***Adobe Premier***
- ***Macromedia Director and Shockwave***
- ***Office Suite***
- ***Lotus Suite***

[redacted]

Photoshop is installed by AIM and executed from estream sets on App server

- ***No Subscription***
- ***No License Management***
- ***RAM cache for ECM***
- ***Installation of Photoshop using AIM***

[redacted]

Photoshop is installed by AIM and executed from estream sets on App server

- ***No Slim Server***
- ***Disk based cache for ECM***

Estream includes initial prefetched pages and these pages are prefetched during installation
Fully functional estream bits (includes initial prefetched pages)
Client software is run as a service
App Server is started by Monitor
Admin UI to stop and start app Server
Application subscription from web server
installation on client after subscription

[redacted]

Testing environment is setup (configuration of 3 servers and one client)

Photoshop runs with the following additional functionality

Leads for milestone: Amit and Nick

Slim Server
http protocol
CNI supports uique message ids for NAD
Fully functional LSM
Real Accesstokens
Uninstall applications
Anti-Piracy support
AppServer and SlimServer fail-over
File spoofing

[redacted]

Clean builds by integration (George) **(Raj will drive this)**

[redacted]

Office is running with full functionality

Restructuring of client so it can be started at boot time
Performance tuning
Improve robustness
application upgrade
Crash resiliency
All software purified and memory leaks eliminated
(May be) Applets for monitoring server components

[redacted]

Office is removed from desktop of at least one person and reinstalled using estream

[redacted]

Code Freeze

Engineer

[redacted]

Server
Sameer
Mike
Bhaven
Amit
Jae Jung
Chungying Chu

Builder

David
Bob
Sanjay

Client

Curt
Anne
Nick
Raj
Ameet

[redacted]

THIS PAGE BLANK (USPTO)

eStream 1.0 High Level Design

Version 1.0

[redacted]

Introduction

This document describes the high level design for the eStream 1.0 product. It is essentially a summary and a tying together of the low level designs for each component in the system. The organization of this document is:

- Basic overview of the entire system
- Block diagrams for the client, server, and builder portions, showing all major components
- General discussion of each component, and pointers to the low level documents for these components
- A list of known issues

To understand the problem being solved in this design, see the "eStream Requirements Document" for information.

Note that this design is for a Windows NT4.0 and Windows 2000 client **only**. As work progresses on a Windows 95/98 client, the designs here will be updated.

Overview

eStream 1.0 encompasses the following basic features:

1. A distributed file system for application files, residing on a server and cached on a client.
2. A small client "player" program to allow local execution of applications that reside on the servers.
3. Authentication using tokens supplied by a license server to each active client.
4. A managed database of information about applications available to client machines, and subscription and usage data for each registered user.
5. Integration with service provider web servers to allow users to subscribe to apps and manage their accounts.
7. A build system that analyzes applications and enables them to be executed by the client and server.
8. Anti-piracy features to discourage unauthorized copying and use of subscribed applications.

As a way of overview, here are the processes that take place to enable and execute a Windows application from a client machine.

- The eStream builder is used to create an *eStream set* for the application. The application is installed on a clean machine, with the builder tools running. These will monitor all file installs and registry updates required to run the application, and encode them into a binary file—the eStreamSet—that will be installed on a service provider's eStream application server (app server).
- A user must download and install the eStream client (ECE) onto her machine, and register as a valid user from a service provider; this will be done using the service provider's web site.
- The user will subscribe to an application from the service provider; a browser module on the client machine will be notified and send a message to the ECE about this event.
- The ECE will communicate with the service provider's eStream license server (Slim server) to verify the newly subscribed app and all permissions, and will install a small portion of the application onto the client system—essentially, the registry entries, shortcuts, and small shared files necessary for execution.
- All application files that are not installed on the client will be accessed via a separate eStream file system (EFSD)
- The user will now see standard shortcuts for subscribed applications, exactly as though the app were installed locally.
- Starting an application, via a command line or double-clicking a shortcut, will cause the client machine to start executing the application on the EFSD. This means the virtual memory manager will request pages from the EFSD during page faults.
- These requests will be forwarded from the EFSD to the eStream cache manager (ECM), a component of the ECE, and on to the app server, assuming the page requested is not in the cache.
- Before any page request is fulfilled by the ECE, the client license subscription manager (LSM) will check that the user has permission to run the application, requesting an *access token* from the Slim server if an existing one has expired.
- This valid access token is sent from the client to the app server for every page request; this authenticates the request.
- The server monitor will be continually checking the state of the app servers and Slim servers. If any are down, it will take them off line.
- The client has a list of valid Slim and app servers for each registered service provider and subscribed application. If response time for any of these is bad, it will stop using it and fall back on the rest.

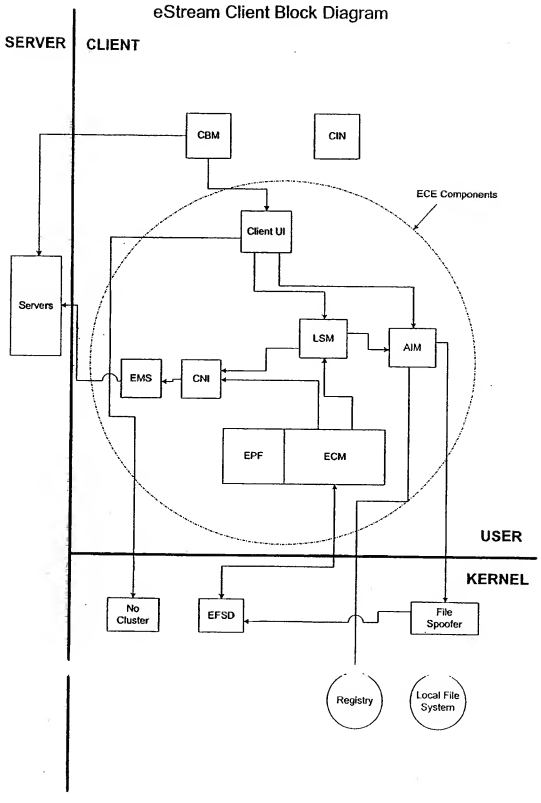
Block diagrams

The following are simple block diagrams of the client and server components. Some conventions:

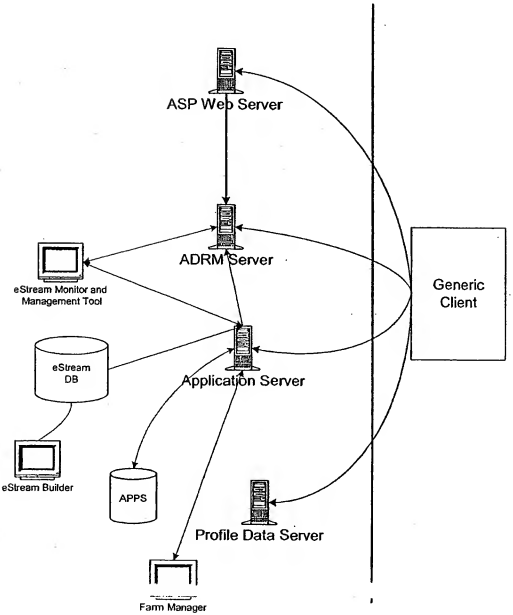
- A box represents a **logical eStream component**. A component may exist as a distinct process, or it may be grouped with other components into a common process.

- A line between components represents an interface call from one to another. If A calls B, there's a arrow on the end of the line at B. If A and B call each other, there's an arrow on both ends of the line.

Note that data stores are **not** represented in these diagrams; if a data store is centrally managed, then there is a component that has interfaces to allow access to these data.


eStream Client Block Diagram
SERVER
CLIENT
CBM
CIN
ECE Components
Client UI
Servers
LSM
AIM
EMS
CNI
EPF
ECM
USER
KERNEL
No Cluster
EFSD
File Spoofer
Registry
Local File System

eStream Server Block Diagram



eStream Builder Block Diagram

???

Component descriptions

Client components

The eStream client consists of the following components illustrated in the diagram above:

- ECE: the eStream Client Executable. This is the aggregation of several user space components into a single executable, operating as a Windows service.
- LSM: the License Subscription Manager (part of the ECE). This tracks and handles all required user information needed by the client: service providers, subscriptions, and access rights.
- AIM: the App Install Manager (part of the ECE). This is responsible for installing all necessary bits onto a client machine in order to run a subscribed application. It also uninstalls all local app bits when unsubscribing.
- ECM: the eStream cache manager (part of the ECE). This is the user-space component that handles requests from the EFSD, and manages the on-disk and in-memory cache of file contents.
- EPF: the eStream PreFetch component (part of the ECE). This works closely with the ECM to handle prefetches of pages for running eStream applications (as opposed to demand fetches, handled by the ECM).
- CNI: the Client Network Interface (part of the ECE). This manages queues of requests from various client components to the app and Slim servers.
- EMS: the eStream Message Service (part of the ECE). This library, used in both the client and servers, handles the actual network sends and receives between remote machines.
- CBM: the Client Browser Module. This is a client-side web browser plugin that is used to handle notification from a service provider's web server to the ECE, when user updates have taken place.
- CIN: the Client Installer module. This small component installs, upgrades, and uninstalls all the required client software.
- FSP: the File Spoofer. This is a kernel-mode driver that is used to redirect requests, intended for local filesystems, to the EFSD. It is a file system filter driver that sniffs all Create requests to the necessary local FSDs, compares the filenames with a list of files that must be spoofed, and if a match is seen, redirects the request to the EFSD.
- EFSD: the eStream File System Driver. This is a standard Windows NT FSD, handling all necessary FS requests from the I/O Manager. It ultimately sends these requests to the ECM to be satisfied (either locally or remotely).
- [illegible] clustering for threads running as part of eStreamed applications.

ECE

The ECE is the Windows service that comprises the bulk of the user-space eStream client software. It provides an overall main program loop, as well as the user interface component for all client components that must communicate with a user.

LSM

The LSM tracks current subscription information and determines the need for license validation. It is informed of subscription changes from the client UI, and is queried by the ECM to validate accessibility to different applications, based on the license model for the subscription to that application.

The LSM has a few major tasks:

1. Keep track of what subscriptions the current user has available from all ASPs
2. Determine which application a given file is a part of
3. Acquire an access token to validate a license for file requests that require one

There are two ways that the LSM updates its list of known subscribed applications:

1. It may be informed of new subscriptions, or of applications that are unsubscribed, by the client UI, as part of a browser plugin in conjunction with an ASPs web site.
2. It may asynchronously poll an ASPs Slim servers to get updated lists of subscribed apps.

AIM

The AIM is the contact point for installation and uninstallation of applications on a client machine. It gets the requests from the LSM to install applications when the user subscribes to them, and it gets requests from the Client UI to uninstall applications.

The AIM manages application installs on the client machine. It keeps track of what applications have been installed on the client machines, where they have been installed and the various components that are part of the installation. It contacts the application servers to get the AppInstallBlock. The AIM uses the AppInstallBlock to then make the appropriate calls to the file spoofer; to install some files on the local disk; to "warm" the cache and to update the start menu and other short cuts as needed.

ECM

The ECM is part of the ECE. Its goal is to:

- Handle all file requests from the EFSD, either by using previously cached contents or requesting the contents from a server.

- Work with the LSM to insure that all applications have appropriately validated licenses before their files are accessed.

The ECM handles the volatile and non-volatile eStream cache on the client machine. It performs demand fetching from the appropriate server(s). Based on the client's observed behavior, it compiles updated profiling data, which may periodically be uploaded to a server.

EPF

The ECM is part of the ECE. Its goal is to intelligently use prefetching of file data to reduce latency of pages requested from the EFSD; this prefetching may result from profiling data or heuristics.

CNI

The client network component is the common point of connection between the rest of the eStream client components and the various eStream servers. Any client module that calls an interface of a server does so through the network component.

EMS

CBM

CIN

The client installer is a simple InstallShield (or simpler) application that will install all of the required client software.

FSP

The purpose of the file spoofer is to redirect file system accesses from some non-eStream drive. This may be necessary in order to support applications running under eStream that are hard-wired to access files in a specific location. The file spoofer may also be used if we are interested in providing a version of some system file different from the one actually on the client machine.

and ensure that these creates are redirected to a file we specify. The redirection could be to a file on the EFSD, or to another, non-eStream'ed file.

EFSD

The EFSD provides standard kernel file system interfaces to the I/O manager and other kernel-mode components. It works with the NT Cache Manager to efficiently cache file

and directory contents. Its view of the ECM is essentially like that of a disk driver, sending primarily read and write requests as needed.

No Cluster

The VM clustering disabling driver (aka NoCluster) disables virtual memory clustering under Windows. While we don't fully understand all the implications, using this driver substantially reduces the average file system paging request size and can dramatically improve performance of eStream, especially on slower connections.

Virtual memory clustering, as implemented in Windows NT/2000, is intended to improve performance when paging to and from physical disks. If possible, we would like to disable clustering only for those threads/processes that will be doing a significant amount of I/O to the eStream file system.

Server components

The following are the server components for eStream 1.0:

- App Server. This is essentially a file server for eStream sets. It satisfies requests for pages from eStream files from the client.
- Slim Server. This handles requests from a client for user and service provider information, and grants access tokens to the client for executing eStream applications.
- Web Server. ???
- Monitor. This enables an administrator to view the server components. It regularly pings the various servers, takes disabled ones off line, and adds new ones to the pools.
- eStream Database. This tracks all user information and server resources for a given service provider.

App server

The application server is there to handle read requests for files accessed by eStream clients. Any file accessed on a client through the EFS can have this read request passed to an app server.

This will be the hardest working eStream server. It will respond to both synchronous [illegible] clients, for many different types of applications and files within those applications.

Slim server

The Software License Management (Slim) server is responsible for:

- Managing data related to users, the groups they belong to, and the applications they are subscribed to
- Validating the licenses for applications executing on clients
- Tracking all outstanding licenses currently in use

ASP web server

This describes only those interfaces on an ASP web server that relate to handling eStreamed applications.

Logically, the ASP web server is the backend web interface for user requests—e.g., get billing information, subscribe to a new app, or request a list of all possible apps a user can subscribe to. In the current model, the web server doesn't actually handle these requests, but instead passes them on to the appropriate eStream-centric server.

Monitor

The monitor utility is responsible for monitoring the overall health of the system. It is responsible to report server status, server traffic, illegal access etc. It will ping the Application Server and the Slim servers to gather the statistics and display them.

Database

Builder components

These are the builder components for eStream 1.0:

- ???

THIS PAGE BLANK (USPTO)

eStream 1.0 Requirements

Version 1.5

[redacted]

1.0 Introduction

This document describes the high level requirements for the eStream 1.0 product. These requirements are given first as lists for the client and server components and then as scenarios.

To facilitate the development of follow-on products, eStream 1.0 does not include attributes that explicitly preclude future support of thin clients or of data ubiquity.

2.0 Client Requirements

The following are performance and functional requirements for the client portion of eStream 1.0:

ID	Description	Priority
1.0	eStream client software operates on Windows 2000, Windows NT4, & Windows 98 for x86 clients.	10
1.1	eStream client software may collect profile information during application execution; this information is used to improve client-based prefetching. This data is not uploaded.	8
1.2	eStream client software supports eStream execution of top-selling (as represented by Ziff-Davis suites) desktop & laptop applications; these applications are listed in section 5.0 below. Other applications are supported (opportunistically) as well.	10
1.3	eStream client software is obtained via the web or via some distribution media & is installed via some industry-standard [e.g., installshield] mechanism; its installation requires administrative privileges. Install reboot should be avoided, unless needed to [illegible] eStream client software can be upgraded w/o reinstallation and w/o breaking installed apps.	10
1.4	eStream client software operates across the different natural languages supported by Windows. The first release is an English-language release.	8
1.5	Applications running under eStream have an ave	8

[redacted]

	interactive response time within 10% of client native for connections at 256K bps or higher.	
1.6	eStream client software is able to operate with only 16M of available disk space; this is the minimum supported configuration. User is encouraged to allow cache to grow beyond an arbitrary limit for best performance.	8
1.7	eStream client software supports simultaneous execution of multiple eStreamed applications, including multiple instances of a single application (by a single user at a time; please see 1.21).	10
1.8	eStream client software is able to unambiguously reference a particular ASP license for an application.	10
1.9	Applications being eStreamed function in the same way that they would if they were installed locally.	10
1.10	eStream client software tolerates server failure [i.e., it continues running any active apps and allowing apps to be launched], though possibly with some delay, assuming that an alternative server of the needed type is accessible.	10
1.11	eStream client software detects and tolerates lost or garbled messages.	10
1.12	It is difficult to steal an eStream application's code or data from the client.	10
1.13	When an eStream application is being installed on a client, the process detects if the app is already installed & requests user confirmation to continue; a single version of an application is available to the user at a time. Install reboot should be avoided, unless needed to minimize potential stability/reliability problems.	10
1.14	Upon uninstalling an application, application-specific changes to the client system are removed or undone.	9
1.15	eStream client software makes minimal changes to the client system when running, avoiding/hiding any registry, DLL, & non-Z file system changes as needed.	10
1.16	eStream client software makes a run/no run license decision quickly enough when an eStreamed application is started not to cause customer satisfaction issues.	10
1.17	eStream client software is launched/terminated at boot/shutdown. It is normally activated/deactivated at logon/logoff of an eStream user; it may	10

	temporarily be deactivated/activated at other times to allow specific administrative activities to occur.	
1.18	User is able to set initial size of client cache. User is able to increase the size of the cache later without significant performance penalty.	9
1.19	eStream client software does not include explicit ASP logon/logoff to run installed apps; ASP identification data stored on client machine allows AccessToken to be obtained in seamless manner.	9
1.20	eStream client software facilitates roaming (i.e., moving one's client system to a different site); eStream server info is not invalid/inappropriate when the client is moved to a different venue.	10
1.21	Only one user at a time on a particular client can have eStream active.	10
1.22	eStream will not allow users to run the same application from multiple clients simultaneously if the license prohibits it.	10
1.23	eStream client user interface will support HELP and ABOUT functions, including links to websites with FAQs and support access information	10
1.24	To start the process of having a particular ASP's application subscriptions known & kept updated on a client, one must visit the associated ASP website with that client at least once.	

3.0 Server Requirements

The following are performance and functional requirements for the server portion of eStream 1.0:

ID	Description	Priority
1.0	eStream provides user and account management capabilities.	10
1.1	User Account Creation/Deletion supported.	10
1.2	User Account is able to subscribe/unsubscribe to Applications.	10
1.3	User is able to view billing and account information.	10
1.4	User is able to change password/address/billing information online.	10
1.5	User is able to list all available & subscribed	10

	applications.	
1.6	User is able to access online help/doc, including an FAQ database.	10
1.7	Omnishift provides interfaces to facilitate customer support by third parties.	10
1.8	User is able to enter/modify data securely.	10
1.9	Both IE 4.0/later and Netscape Navigator 4.0/later browsers are supported.	9
1.10	ASP agent [i.e., special administrative user at the ASP] is able to access all user information.	10
1.11	ASP agent is able to disable a user.	10
1.12	ASP agent is able to modify license information for a user.	10
1.13	User is able to add additional users to the account.	10
1.14	Web Server is highly scalable.	10
1.15	Servers are able to operate in non-English language.	9
1.16	ASP may operate eStream system with single server.	9
1.17	Flexible access/export of billing information is supported, to facilitate 3rd party billing systems.	10
1.18	eStream server software and eStream apps can be upgraded w/o impacting installed eStream client software. All upgrades are backwards compatible.	10
2.0	The eStream framework [ASLM Server] provides a mechanism to validate the usage of application components with respect to billing models.	10
2.1	ASLM server is able to validate users to use specific applications.	10
2.2	ASLM server records all usage activity down to the granularity necessary to support billing models. The granularity will be reasonably large.	10
2.3	ASLM is able to release license on explicit request or timeout from the client.	10
2.4	ASLM is portable across a wide variety of platforms and operating systems, including but not limited to: Windows NT4, Windows 2000, Solaris UltraSPARC, and Linux.	10
2.5	ASLM servers are fault-tolerant.	10
2.7	ASLM server is able to report Denial of Service attempts.	10
2.8	ASLM server reports illegal accesses.	10
2.9	ASLM is able to register its presence/load to the Web Server(s).	9
3.0	eStream Framework provides management and monitoring tool (EMMT) to manage the servers.	10

3.1	EMMT is able to start/stop servers in the eStream framework.	10
3.2	EMMT is able to monitor server activity for all servers in realtime.	10
3.3	EMMT is able to configure the servers.	10
3.4	EMMT is able to provide historical reporting.	9
3.5	EMMT is able to display information graphically and in spreadsheet format.	8
3.6	EMMT is able to raise alarms on predefined events.	9
4.0	eStream framework provides a mechanism to deploy the application via the eStream Builder.	10
5.0	eStream framework support a variety of licensing models.	10
5.1	Floating license model is supported. n User – k Licenses	10
5.2	Names User License model. (Special case n=k)	10
5.3	Time based licenses at billing granularity.	10
5.4	High water mark license.	10
5.5	Node locked licenses.	8
6.0	App Server is able to Authenticate client's accesses (via AccessTokens) completely locally.	10
6.1	App Server encrypts returned data (via a random key chosen by the client); it must be computationally infeasible to steal an application's code while it is being distributed or to determine which application a client is running.	10
6.2	App Server is as stateless as possible to allow client to switch to alternative app server w/o significant overhead. "Stateless" means that there is no server context that would be lost if the server went down; one classic example of this is that "file open" is recorded on the client, not on the server.	10
6.3	App Server is optimized to respond to requests with minimal server load, thereby maximizing scalability.	10
6.4	App Servers may be grouped along with any number of other such servers into a farm with minimal inter-server interactions (as to maximize	9
6.5	App Server communicates with clients thru firewalls.	10
6.6	App Server communicates with clients efficiently (e.g., via persistent HTTP connections).	10
6.7	App Server is able to install new eStream sets w/o having to go down.	10
6.8	App Server is robust, able to run for long periods	10

	without crashes (i.e. no resource leaks, and handles most/all failure modes for system operations); 24/7 operation.	
7.0	App Servers, ASLM Servers, ASP Web Server and EMMT communicate through a database which will include but need not be limited to Microsoft SQLServer.	10

4.0 Builder Requirements

The following are performance and functional requirements for the builder portion of eStream 1.0:

ID	Description	Priority
1.0	The Builder installation monitor runs in the background, when an eStream application is installed as part of its preparation or building capabilities.	10
1.1	The Builder installation monitor captures all the updates to the System Registry that take place during the install.	10
1.2	The Builder installation monitor records all the files created in the two kinds of directories: the install directory and the common directories.	10
1.3	The Builder must be able to gather initial set of application profile data. This data consists at least of the page access pattern for starting and immediately shutting down an application	10
1.4	The Builder must package the eStream Set into an easily manageable packages suitable for ASP administrators to download to their servers.	10
1.5	The Builder must be able to collect per-user profile data from the Profile Server and merge the profile data into a combined data usable for updating the profile data in the appInstallBlock.	8
1.6	The Builder should be run in an environment where no other applications are running.	10
1.7	The Builder should provide functionality to create installation set(s) for each of the clients eStream 1.0 is going to support.	10
1.8	It should be possible to change the appId of the eStream set when an ASP wants to "install" the eStream set in order to host it.	10
1.9	It should be possible to create a merged eStream set	10

	for a suite of applications.	
1.10	It should be possible to test the eStream Set created by the Builder using a stand-alone tester and not require the eStream client+server programs.	10
1.11	The appInstallBlock should have support for indicating upgrades at the support site	10
1.12	In the process of creating an eStream set it should be possible for the user to delete file entries and registry entries manually to "trim" the eStream set if she so desires assuming the user knows what she is doing.	10
1.13	The Builder should be run in a clean machine with as few software installed/upgraded as possible.	10
1.14	The Builder should support individual applications in a suite even if the installer of the suite doesn't allow installation of individual applications.	10
1.15	The Builder must be able to create an initial set of cache contents for the eStream client and allow the initial size to be selectable by the user or automatically.	10

5.0 Client Use Cases

5.1 USE CASE: Installation of eStream client code

- Obtain eStream client code bits.
- Install z: file system hooks & setup to have z: mounted at appropriate time.
- Install eStream client code, which services z: file sys requests from local cache or from servers & which handles sideband communication w/ servers, and setup to activate estream client code at time desired by user (boot, login, on demand).
- Install NoCluster.sys to disable page fault clustering at system boot.

5.2 USE CASE: Installation of application

- Obtain AppID & App Server name for installation from SLM Server.
- Download AppInstallBlock information.
- Perform initial installation & setup for app, after checking system for previously installed version of app & issuing any appropriate warnings.

5.3 USE CASE: Uninstallation of application

- Remove all registry/DLL/filesys changes associated with app installation.
- Remove all other data associated with application.

5.4 USE CASE: Uninstallation of eStream client code

- Remove z: file system hooks, eStream client code, & nocluster.sys.

5.5 USE CASE: Execution of eStream client code

- Respond to z: file sys requests and detect when new eStream app is referenced.
- Support Client UI requests.

5.6 USE CASE: Execution of application

- Obtain Access Token & list of App Servers from SLM Server.
- Contact App Server(s) as desired to obtain file system data.
- Respond to running application's requests, collect usage data. Cache portions of application, file system info, & user preference info.
- Detect server connection issues (apparent loss of connection or connection response below acceptable threshold) & licensing issues; negotiate with ASLM Server as needed.

6.0 Server Use Cases

6.1 USE CASE: Create an Account

- Customer brings up browser and connects to ASP Web Server
- Screen display shows "create account", customer selects and enters required account info (billing info, UserID, pword, etc)
- ASP Web server writes account info to Acct DB using **AddAccount** where a unique Account ID is assigned
- Account ID is returned via web page

6.2 USE CASE: Create a User

- Customer brings up browser and connects to ASP Web Server
- Customer enters their userid and pword
- ASP Web server contacts Acct DB, using **AddUser** userid and initial password, gets Acct info and displays to Customer
- Customer selects "add user" and enters required user info (username, address, email etc)
- ASP Web server writes user info to Acct DB updating account info

6.3 USE CASE: Modify Account

(includes disabling an account or user, removing users from accounts, changing pwords etc)

- Customer brings up browser and connects to ASP Web Server
- Customer enters their userid and pword
- ASP Web server contacts Acct DB, passes along userid and pword, gets Acct info and displays to Customer
- Customer selects "update info" and enters desired changes
- ASP Web server writes updated account info to Acct DB

6.4 USE CASE: AddSubscription

- Connect to ASP web server
- Enter account number, username, password
- Verify that user is account admin using **GetUserPermissions**
- Get list of possible subscriptions (using **ListPossibleSubscriptions)**
- Get list of current subscriptions for account (using **ListCurrentSubscriptions)**
- Display in page – User chooses a subscription and license type
- Display a screen to allow the user to configure the license. For a floating license, allow selection of users, etc.
- Call **CreateSubscription** to compose the new subscription for each user and create licenses.

6.5 USE CASE: Building an eStream set:

- Start w/app **CD-ROM**, and a freshly installed OS (plus latest service pack?).
- Install app into Z: drive (could just be a regular network drive)

- A **special system monitor** logs all registry changes and file system changes during the install.
- File system changes to C: during install probably need to be spoofed (or have a registry entry point to Z: instead), especially newly added directories, so need to do the appropriate thing.
- From this log and the actual files as installed on the machine, the **eStream set builder** creates the eStream set, which is a small set of related files.
- Separately, we need to actually set up the app for eStreaming, then run it and collect profile data to seed the initial page prediction map.

The **AppServer UI** (interface to user to control an Application Server on a particular machine) presents the following management functions:

A. Starting a server:

- **AppServer UI** always indicates whether an AppServer process is up and running (and alive w/status), and if present prompts for restarting the current server process.
- Otherwise it goes ahead and starts up the AppServer process and reports any errors.

B. Stopping a server:

- Simple, just stops any running servers, gracefully, perhaps prompting user for ungraceful shutdown if not successful.

C. Install eStream set:

- Each server is configured with a specific eStream set directory, under which it places (in their own individual directories) the actual eStream set contents (a few files on the native file system).
- User indicates to **AppServer UI** where to find the eStream set package provided by Omnishift. **AppServer UI** authenticates the package, and verifies its integrity, and if successful, unpacks and places the constituent files in the server's eStream set directory.
- Note that it is possible for the eStream set directory to live on a file server shared by other **Application Server** machines, so installation may be required only once

  is responsible for replicating eStream sets across a farm to ensure the farm machines are symmetric).
- How does the server know a new eStream set is available? Each set is assigned a VolumeID, and the set contents can be placed under a directory with the same name as the VolumeID. The install is synchronized via a AppList file, which just lists the valid VolumeIDs, which the **Application Server** only reads, so an entry is added at the end of the install procedure. The **AppServer UI** then must send

some kind of message/signal to the Application Server to have it resync with the file (and start serving the new app).

- Also note that eStream set install is doable without bringing down the server (or any server in the farm.
- Having done this, it will probably be necessary to notify the **SLM** and **Account Servers** that a new app is available. With some scripts provided by omnishift, this sould be done by a human administrator. They need to know the VolumeID of the app that was installed along with the full name, so that the client can initiate an app install procedure via the VolumeID (the server can then provide the **AppInstallBlock** which probably has a fixed reserved global FileID).
- Questions: What if app is already installed (want to allow reinstall or force remove first)? What if app is being upgraded (probably also should be a remove and then install)?

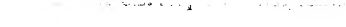
D. Remove eStream set:

- First we probably have to disable the application on the **SLM/Account servers**.
- This probably will require sending some kind of message to the Application Server (if running) to stop serving the given eStream set, and then waiting for any active connections to expire.
- Then we can just remove the entry in the AppList file, and delete the file system image.

E. Configure Application Server:

- The **AppServer UI** presents various configuration options to the user (stuff like logging, port #, threads, etc.) Some may require restarting the **Application Server** to take effect, others may take effect immediately.

Another activity that occurs, automatically, is the processing of profile data. It is not clear what the page prediction map looks like, but clients will periodically send profile data to the **Application Server**, which aggregates it, and must store it persistently, and to allow new clients to benefit from improved prediction. There may need to be a special module that can take the aggregated profile data to modify the prediction map.



Where are we?

- Client PC has installed the subscribed app and has received a subscription token, and the name/IP of SLM Server.
- Customer is accessing an app file and doesn't have an access token for it, yet. (i.e. double clicking z:\word.exe).

Players involved: client – cache mgr, SLM Server and indirectly, User/Account/Sub/Rights DB.

What happens:

- Client contacts the SLM server and gives: subscription token, user/passwd.
- SLM server looks into the user/acc/sub/right DB to
  - Authenticates user and password; may return: "invalid user".
  - Authenticates subscription token; may return: "invalid subscription token"
  - Look at the Accounts container and see if any licenses are available. If so, check it out by creating a new access token and updating the accounts container. It may return: "can't get license".
- Return an access token to the client and a list of app servers.

6.7 USE CASE: Process File Request – steady state

Where are we?

- Client has installed the app and has a list of app servers,
- Client is holding a valid access token that it acquired from the SLM server.
- Client, while processing an IRP, needs to access portion of file on the app server.

Players involved: client – cache mgr, app server. NO SLM server or no user/account/dub/rights DB.

What happens:

- Client contacts one of the app servers and gives: access token, App ID, File ID, length and file offset.
- App server quickly verifies the expiration date on the access token.
  - It must not need to contact the user/account/sub/rights DB to do this. It only cares about the time-validity of the token. If token has expired, return some kind of an error back to the client.
- App server locates the data and sends it back to the client.

NOTE: we are simplifying this quite a bit when discussing the scenarios because we are not sure exactly how we are going to

manage the server farms. Another key question is MUST all app servers host all estream apps ?

6.8 USE CASE: Renew an Access Token – steady state

Where are we?

- Client acquired an access token from the SLM server.
- While running the app, client sees the needs to renew the access token. This may happen synchronously when the user touches one of the app files, or by a timer-driven client daemon that periodically renews an access tokens before it expires.

Players involved: client – license manager, SLM manager, and indirectly, user/accounts/sub/right/ DB.

What happens:

- Client sends an access token to the SLM server.
- Check the time-validity of the access token.
  Assumption: SLM server assumes that only valid access tokens can be renewed. An expired token implies a lack of renewal, which implies releasing the license. SLM server can try to acquire the license, but there is no guarantee that it will succeed.
  - If token is expired app, goto Scenario: Acquire Access Token.
- SLM server accesses the user/account/sub/rights DB to:
  - Generate a new token that will expire some time in the future (configurable parameter).
  - Update the account container in user/account/sub/rights DB.

Return the new access token.

6.9 USE CASE: Validate user request for access to an application server

Procedure:

Query AccountDB for license to access application appID in subscription subID
If (no valid license) then
 Send FailureReason to Client
Else
 Send accessToken, appServers to Client

6.10 USE CASE: Add subscribable application from an account

Interface Required:
SLMServer::AddSubscribedApp(accountID, subID)
Procedure:
Receive accountID, and subID from the Client
Check for valid accountID, and subID on AccountDB
If (no valid accountID or subID) then
Send FailureReason to Client
Else if (subID is not already subscribed under accountID)
Add Subscription subID to Account accountID in AccountDB
Send Success to Client

6.11 USE CASE: Remove subscribable application from an account

Interface Required:
SLMServer::RemoveSubscribedApp(accountID, subID)
Procedure:
Receive accountID, and subID from the Client
Check for valid accountID, and subID on AccountDB
If (exist subID in accountID) then
Remove Subscription subID from Account accountID in AccountDB
Send Success to Client
Endif
Send FailureReason to Client

6.12 USE CASE: Monitor/management tools

Interface Required:
SLMServer::GetTrafficHistory()
SLMServer::GetUsageInfo(userID, appID, subID, accountID)
SLMServer::GetCurrentTraffic()
SLMServer::AddServer(serverID)
SLMServer::RemoveClient(userID, serverID)
SLMServer::GetErrors()
SLMServer::DumpErrors(filename)
SLMServer::DeleteErrors()
AppServer::GetTrafficHistory()
AppServer::GetCurrentTraffic()
AppServer::GetErrors()

Procedure:
SLM Servers keep track of traffic info. The monitor/management tool can query the SLM/App Servers anywhere for traffic info. Some examples of traffic data:
- Traffic history of particular server on number of clients served per unit time
- Monitor length time a userID used application appID under subscription subID and charged to accountID
- Monitor current load information on all servers (SLM server and app server)
- Allow admin manually add/remove some servers from the pool.
- Allow admin to kick some clients off the server.

The monitor/management tool can also be used to display a list of errors logged by the servers.
- Monitor errors and be able to categorize by error type
- Monitor errors occurring between certain time periods
- Monitor errors reported by a particular server
- Manage errors to dump the errors to a file
- Manage errors and delete a subset of errors

Finally, the monitor/management tool can check for any illegal accesses.
- Monitor failed attempts to access SLM Server with bad password, especially on repeated failed attempts in a short time frame.
- Monitor any attempts to use a particular license and failed.
- Monitor access to SLM Server from non-typical IP addresses for a particular account. The server is required to save the history of IP addresses of accesses to a particular subscription account.

6.13 USE CASE: Adding a new application server.

Summary:
An application server's functionality is to provide applications eStream sets to client application. An application server is generally added to the system to provide greater scalability and/or to provide additional application support.

Actors:
1. [illegible]
2. ASLM Server(s): The ASLM server needs to be notified of the presence of an additional application server and the services it provides.

Inputs:
1. Application server(AS) installer
2. Application eStream sets. These may be available from one of the following location: AS installer, some other AS or Farm Manager Server(some central repository) .

3. SLM Server location(This input may not be required based on scalability solution that we decide on).

Processing:

1. Using the AS installer install the application server.
   a. <Server install use case to be added here? Later>
2. Copy the Application eStream sets. There are several options here:
   a. Provide the eStream sets as a part of the installer.
   b. Provide a script to ftp to another Application server and copy the eStream sets.
   c. Provide a management tool to manage the copying of the eStream sets. From the ASP's perspective this is the best solution. A tool which provides tracks the application would be useful to manage the load.
3. Configure the server. The server needs to know the additional application sets that it supports(? This may not be required).
4. Start the server.
5. Register the server with other SLM servers. The following options apply:
   a. Multi-cast the "new server and services" message to the SLM servers.
   b. Register the server to a local object server which in turn notifies the object servers across the system. CORBA model supports this.
   c. Using the resonate model(described below), all appservers are essentially the same server. ie Address app.foo.com will point to a set of app servers. A new server enabled will resonate software will automatically register itself with the resonate scheduler. (How do we make the resonate scheduler aware of the applications available on the app servers?)

Outputs:

1. The App server is installed and running with a set of applications available on it.

6.14 USE CASE: Removing an Application Server.

Summary:

An ASP administrator may decide to remove an application server from the system for various reasons. Removal of server from the system would result in notification to the rest of the SLM servers that it is no longer available for servicing the objects.

Actors:

1. ASP administrator
2. SLM Servers.

Inputs:

1. Application server running on the machine.
2. ASLM Server(s): The ASLM server needs to be notified of the presence of an additional application server and the services it provides.

Processing:

1. Stop the application server. This will result in the Application server informing the rest of the ASLM servers that it will no longer take any requests. This in turn may result in an application being unavailable for usage. Depending in the framework used, this can be done in one of the following ways:
    a. Multi-cast the message to the ASLM servers.
    b. Just stop the server in the CORBA framework. The local ORB server will notify the unavailability of the resource to the rest of the framework.
    c. Using the resonate model to scale would imply that you just stop the server the resonate agent on the server will notify the resonate scheduler to deregister the servers. (However its not clear if you can also deregister the objects served by the server.).

Outputs:

1. ASLM servers are notified of the removal of the resource.

6.15 USE CASE: Add a new ASLM server.

Summary:
The ASP provider may decide to add an additional ASLM server to enhance the performance of the system. The additional ASLM server added to the system should be accessible to the ASP's Web Server so that it can direct the clients to the SLM server. (This may not be required if we deploy the Resonate model of scaling).
Actors:

1. The ASP administrator.
2. The ASP Web server.

Inputs:

1. ASLM installer
2. Web Server location(This input may not be required based on scalability solution that we decide on).

Processing:

1. Using the ASLM installer install the application server.
   <Server install use case to be added here? Later>
2. Start the server.
3. Register the server with ASP Web Servers. The following options apply:
    a. Multi-cast the "new server and services" message to the Web servers.
    b. Register the server to a local object server which in turn notifies the object servers across the system. CORBA model supports this.
    c. Using the resonate model(described below), all ASLM are essentially the same server. ie Address aslm.foo.com will point to a set of ASLM servers. A new server enabled will resonate software will automatically register itself with the resonate scheduler.

Outputs:
The ASLM server up and running.

7.0 Builder Use Cases

7.1 USE CASE: Install Monitoring

- Query builder for CD media and installation executable(s)
- Monitor various registry and file updated during installation
- Merge installation data for all applications in a suite
- Relocate files from C: to Z: directory
- Create appInstallBlock and package the appInstallBlock with the application files

7.2 USE CASE: Profiling

- Query builder for application executable(s)
- Monitor sequences of file accesses from OS to the file system as profile data
- Identify the subset of the profile data as the initial cache contents
- Merge profile data and initial cache contents into the corresponding appInstallBlock

8.0 Key Applications for eStream 1.0

Winstone99:
Business: QuattroPro, WordPerfect, Lotus® 1-2-3, Word Pro, Access, Excel, PowerPoint, Word
High-end: Adobe® Photoshop, Adobe® Premiere, Microsoft® FrontPage, Sonic Foundry® Sound Forge
Content Creation Winstone 2000: Macromedia Director, MacromediaDreamweaver

Please note that release of Business Winstone 2000, which was originally slated for 6/27/2000, has now been postponed until the Fall Comdex & will be called Winstone 2001. As soon as the contents of this suite are released, we should move quickly to assess our support for its application set.

THIS PAGE BLANK (USPTO)

eStream 1.0 Server Scaling Estimate

Anne Holler [redacted] *Version 1.0*

Introduction

This document presents an estimate of server scaling for the eStream 1.0 product as compared with its chief competitor, the Citrix product as deployed by Personable. The document presents relevant attributes of the basic application execution model for each of the two products, discusses and gauges the impact of the areas in which server scaling differs between them, considers the effects of additional attributes of the two products on server scaling, & finally summarizes the differences in expected server scaling in terms of a number. Please feel free to challenge the assumptions, methodology, & calculations herein, now & as we move forward through the design & implementation phases.

In the process of developing this server scaling estimate, certain assumptions about user, system, & program behavior are made, & certain design/implementation goals of the eStream 1.0 product are assumed. This material is listed in separate sections at the end of the document for ease of reference.

This work does not intend to imply that the user experience of the eStream 1.0 & Personable/Citrix products is expected to be comparable with respect to the relative server scaling point identified. Interactive response differs between the two products; first-hand experience with server-based applications running on New Moon & Personable/Citrix and client-based applications running on the eStream prototype suggests that the former are sluggish on an ongoing basis with respect to activities such as selecting from pull-down menus & the latter are as responsive as native wrt such activities, with noticeable delays engendered only when heretofore unused portions of the application's functionality are exercised. Reliability also differs between the two products; smooth fail-over of an active Personable/Citrix application to another server is not supported, whereas such fail-over is included in the eStream 1.0 design.

This document does not address an area in addition to server scaling that may be of competitive interest to ASPs; that area is network bandwidth differences between eStream 1.0 & Personable/Citrix. Though it might seem intuitive that sending application pages across a network consumes more bandwidth than sending user input & display output, aggressive client caching ameliorates the traffic associated with application paging, whereas the traffic associated with the display output can be quite substantial according to Harwood's book "WIN Terminal Server & Citrix Metaframe" [hereafter, HARW99]. It may be worthwhile to collect and compare bandwidth data wrt the two products.

Acknowledgements

Amit Patel provided key insights. Any mistakes/bogosity are mine.

Application Execution Models

The following are basic attributes, with respect to application server scaling, of the application execution models of Personable/Citrix and eStream 1.0:

Personable/Citrix Client/Server Application Execution Model
Application executes on server
On app page faults & app data reads, page loaded from server disk
Server handles incoming network traffic for all keyboard & mouse events
Server handles outgoing network traffic for bitmap display update

eStream 1.0 Client/Server Application Execution Model
Application executes on client
On app page faults & app data reads, page loaded from client cache, excpt miss to server
Server handles incoming network traffic for client cache misses
Server handles outgoing network traffic for client cache misses

Application Server Scaling Comparison

The impact on server scaling of executing applications on the client, rather than on the server, is expected to be large. Processor & main memory overhead are associated with executing applications, including the overhead for fielding interrupts such as mouse & keyboard events. According to HARW99, processing power for running applications is typically the biggest Citrix product bottleneck, and that is reinforced by the variance in Citrix scaling numbers reported wrt Personable [Ernie] and wrt another ASP [Amit], for which the main differences seem to be application execution overhead. HARW99 indicates that Citrix scales at 10 to 45 applications per processor, largely due to execution overhead (though some overhead is due to processing page faults & accessing application data, which is considered in the next paragraph). It is somewhat difficult to understand how to model the relative benefit for this difference (in the sense that an *infinite* number of applications can run on a processor that they are not actually using to execute!); it seems conservative to assume we get at least as much benefit from this factor as we do from reducing cache miss overhead on the server, so let us have this factor double whatever benefit we project from the factor considered in the next paragraph.

The impact on server scaling of processing application page faults & application data reads on the client in most cases, rather than on the server, is expected to be measurable. For the purposes of the server scaling estimate presented in this document, let us assume that the server overhead to fetch a page from disk, whether the request came from server execution or from client request, is comparable. (Although we can construct file server technology in which client requests take less time than server requests, let us assume that the overhead to encrypt the response consumes that time savings). Also, let us assume that the same number of page faults occur on the client & on the server (which server partitioning for Personable/Citrix could render untrue.) Hence, the difference in server

overhead for application file accesses is estimated to be equivalent in scale to the reduction in the number of references, which is derived from the client cache miss rate. Assuming a client cache miss rate of 2%, each eStream application server can handle 50 times as many clients with respect to this attribute of server overhead. Doubling that amount due to the factor described in the previous paragraph, we estimate that each eStream application server can handle 100 times as many clients as each Personable/Citrix application server.

It is somewhat difficult to know how to compare the network interface overhead component of server scaling between eStream 1.0 and Personable/Citrix. The loading constraints associated with executing applications on the server are expected to limit the amount of network overhead presented to a Personable/Citrix application server. Depending on the kind of application, significant traffic is generated to support client displays, but HARW99 identifies network bandwidth overhead [discussed in the Introduction section] – not server overhead - as the scaling problem engendered by this traffic. Each eStream 1.0 client generates little server network overhead, but the reduction in server load due to client application execution allows more clients to be connected to a given server, possibly straining the network-oriented portions of an eStream application server. At this point, let us assume that server network interface overhead does not materially impact the relative server scaling of the two products.

SSL encryption can dramatically decrease server scaling; by more than 70%, according to Igor Balabine. He indicates that third party SSL accelerators should be employed to remove this overhead.

Additional Server Scaling Considerations

For eStream, application installation induces load on the application server to deliver to the client the contents of the AppInstallBlock, which may contain registry & file spoofing information, initial cache & profile data, file system structure information, etc. Personable/Citrix does not have a comparable feature. Installation is expected to be an infrequent occurrence and Omnishift is expected to suggest/provide mechanisms to smooth out (or specially handle) high peak demand at particular points in time, including product or application launch/upgrade. Let us assume that this (managed) overhead reduces application server scaling by the equivalent of 0.5% cache miss. Adjusting the running total by this amount implies that an eStream application server can handle 67 times as many clients as a Personable/Citrix application server can.

does not have a counterpart on Personable/Citrix [except for page fault clustering ;-) !]. Given that eStream 1.0 is targeted at fat clients, client prefetching (which is redundant wrt a comparably warmed client cache) is expected to be used sparingly. Let us assume that prefetching adds the equivalent of an additional 0.5% cache miss rate; the intuition here is that prefetching is essentially engendered when cache misses occur (i.e., when we are exercising parts of the app we have not exercised before) and that we get unneeded pages a measurable percentage of the time. Updating our running total by this amount

means that an eStream application server can handle 50 times as many clients as a Personable/Citrix application server can.

Both eStream & Personable/Citrix products include several logical servers in addition to the application server. Both have an ASP Web Server portal to an ASP's account services, from which a user can obtain billing information, get a list of available applications, subscribe to new applications, etc. For both, the ASP web server interfaces in some way with an account database of presumably comparable complexity. Both eStream & Personable/Citrix have server functionality involving getting the license to run an application, which is expected to cause similar database overhead; in eStream 1.0, this process involves getting an AccessToken from an ADRM server. However, Personable/Citrix does not have eStream 1.0's concept of renewing an AccessToken. It is expected that the eStream 1.0 design will take special care that renewal does not add significant overhead to the eStream 1.0 ADRM server (by specifying nontrivial billing granularity & AccessToken renewal frequency, by ensuring renewal is lightweight, perhaps by having some explicit mechanism aside from AccessToken renewal for token cancellation in the event of client failure/disconnect, etc.). eStream 1.0 has the concept of records containing application profile information being uploaded to a server; it is not known that Personable/Citrix has any comparable feature (although the product likely has much other user information recorded at the server, including preferences, etc). It is expected that eStream 1.0 design will emphasize minimal server impact for handling this data. [Profile data may not be a core deliverable for competing with Personable/Citrix.] In summary, it is assumed that server scaling for auxiliary servers is comparable between eStream 1.0 & Personable/Citrix.

Conclusion

Based on the discussions in the previous sections, eStream Server Scaling expected to be significantly higher than that of Personable/Citrix. Considering client execution benefits, client caching benefits, application installation overhead, and prefetching overhead, eStream server scaling is expected to be approximately 67 times higher than the Personable/Citrix competitive product. For some given Personable/Citrix application server that can handle 20 clients, an eStream application server can handle 1340.

Assumptions Underlying Server Scaling Estimate

User's application usage pattern [what is run, for how long, what features] does not change materially depending on whether s/he is using eStream 1.0 or Personable/Citrix. It would be interesting to collect data on typical usage patterns, similar to those estimates for his network bandwidth evaluation, but it might be interesting to see if our in-house use of common applications matches those estimates.

Overall application server capacity is adequate for both products. Personable/Citrix may deny application server access to clients when insufficient overall application server capacity is available, whereas eStream 1.0 may continue to allow clients to access the

servers without some explicit client capacity cutoff point, given the usual small impact of each additional eStream client. However, it is possible that an extreme performance collapse could occur on eStream, if client load were to grow so large compared with the capacity of eStream's application servers that the lack of server response caused an escalating number of redundant requests from eStream clients retrials. This situation needs to be avoided, in the eStream 1.0 design and/or in its deployment.

The increased client servicing capacity afforded an eStream 1.0 application server will not uncover some insurmountable system bottleneck never encountered with the limited client servicing capacity possible on the Personable/Citrix server, including areas such as maximum number of threads, processes, sockets, buffer size for socket send or receive, socket listen queue length, network buffer cache, maximum number of file handles, etc.

eStream 1.0 clients are fat enough to allow adequate client caching to hit or better the client cache miss goal of 2%. Thin clients (which are not the intended design target for eStream 1.0) or fat clients with inadequately sized client caches would increase server overhead for paging requests & network traffic beyond the levels estimated in this document.

Design Goals Supporting Server Scaling Estimate

Overall client cache miss rate is less than 2%. Reaching the eStream 1.0 client performance goals also reinforces the need for a low client cache miss rate. We have not collected data indicating how large a client cache would be needed to hold application pages and file metadata associated with typical application usage as represented by the Ziff-Davis benchmark runs. I think it would be useful to have such data, since it may influence client cache design & effective cache management policies.

AppInstallBlock server overhead is no more than the equivalent of an extra 0.5% cache miss rate. Installation is expected to be relatively rare, & downloaded material is expected to be kept at the minimum size necessary to reach our functionality & client performance goals.

Client wasted prefetch overhead is no more than the equivalent of an extra 0.5% cache miss rate. With fat clients & large warm caches, prefetching is expected to be kept at a minimum for eStream 1.0; again, data gathering related to this area may be useful.

THIS PAGE BLANK (USPTO)

eStream File System Straw Man Proposal

Version 0.5

[redacted]

Purpose

The purpose of this document is to present a concrete proposal for the functioning of the eStream file system. In many places, I make some sweeping generalizations about how things should work without describing the data structures and interfaces involved in implementing them. This document should eventually involve into a design specification.

Issues Not Covered

This document does not attempt to cover all issues present in designing the eStream 1.0 product. In particular, the overall authentication/licensing/security architecture is not covered in detail here. It is expected that the security functionality will be mostly orthogonal to the design of the basic file system functionality.

Background

There are a number of different networked file systems out there. Many of them share some requirements with eStream. For example, AFS performs client-side on-disk caching, while Coda handles serious server redundancy and disconnected operation. Personally, I believe that AFS and Coda are the file systems whose designs are most relevant to us. For those interested in further background reading, you might also want to look at papers covering NFS, CIFS, xFS, DFS, and Zebra.

Single File System Name Space

Many modern distributed file systems present the network file system as a single tree mounted at some location on the client system, regardless of which server hosts the data. (In fact, with AFS, every file on every server in the world can be accessed through a path starting with /afs on the client, assuming the client can reach that server and has sufficient privileges to do so.) Compared with systems like NFS and Windows sharing, where each share is mounted in a different location on the client, the single name space provides greater ease of use.

The eStream file system would present one universal logical file system. Regardless of which ASP provider supplies a particular volume, that volume will always be referenced via the same path on the eStream file system. That this is desirable or even feasible is predicated on the assumption that OTI is the only entity providing all eStream sets. Each volume must get a unique identifier and a unique location to be mounted in the file system hierarchy. If two different ASPs provide the same volume ID, then the contents of those volumes must be identical. This way, we don't have to tag things in the cache based on what ASP they came from, and the cache manager doesn't need to know anything about ASPs. If done correctly, only the client networking component and the LSM need to know about ASPs.

Volumes

A volume is a complete subtree of a file system. Volumes may contain files and directories. Volumes may not be mounted in other volumes. A volume is a logical grouping of files within the file system and is the unit of replication across servers. An application will reside in a single volume. Two applications will never share a volume.

Volumes are uniquely identified by a 32-bit volume identifier. Each volume additionally has an 8-bit version number. This version number is incremented each time any file within the volume changes. (See supporting upgrades, below). Note that the volume id is globally unique. If two ASPs provide volumes with the same volume number (and version), they have identical contents.

A volume may be replicated on any number of servers. Each SLM server contains a map describing the application servers that currently provide each volume. This global replication of this table is acceptable because volumes are added or moved infrequently.

Identifying Files
Files and directories are uniquely identified by the pair (volume id, file number). This tuple is called a file id. Volume id and file number are each 32-bit signed integers. Negative values for both volume id and file number are reserved for special purposes, leaving us with 2^31 possible volume IDs and 2^31 possible files per volume.

Finding an Application Server for a Volume
The SLM will tell the client which application servers currently provide each volume. It may be necessary for the client to periodically poll the SLM to get up-to-date information about the state of the application servers. The License and Subscription Manager on the client will keep track of the currently subscribed applications and the application servers for each of these applications.

Directories
Directories are specially formatted files that are used in a special way by the file system. They are identified by file ids, just like other files. From a client-server point of view, they are read by the client in the same way as other files. Directories contain arrays of entries with the following format:
(volume number, file number, flags, length, filename)

The volume number and file number are 32-bit signed integers. The flags are 32-bits of flags. The length is 16 bits and is the length of the filename in bytes. The filename is a non-NUL terminated Unicode string. The structure is padded with enough Unicode NUL characters to make the structure a multiple of 32 bits long. The next directory entry begins on the next 32-bit boundary.

The access token is not part of the directory, as a single access token is required to access all files in a particular volume.

The volume number is required so that the the client can construct a local directory for the root of the directory structure in the same format as other directories (see filename parsing below). It also helps to provide a sanity check.

Accessing Files
Assuming that a client has a file-id for a file that it wishes to access, the following client-server actions must be supported:

For stat-like information on the file, we need a GetFileMetadata() interface. The client would provide the file id it is interested in and the proper access token for this file. The server will either return the metadata for the file or an error condition (like access token expired or incorrect access token.) The metadata contains the standard Windows metadata information, including file length and file access times.

On a file open (CreateFile in Windows terminology), we need to verify that we have access to the requested file. This is probably best accomplished by calling GetFileMetadata and verifying that we can get the metadata. This way, we can fail file opens gracefully if we don't have an access token.

On reads (and writes, when we support them), the client will send the file id and the access token to the server along with an offset and a length for the read and write. The server will respond with the data. Note that the same mechanism will be used for reading both files and directories.

Pseudodirectories
For those parts of the eStream file system name space that do not belong to any volume (such as the root of the file system), the client must construct appropriate directories based on the currently installed applications. This is to support filename parsing starting at the root of the directory. For example, if the client has word installed with a root of /Worddir and it is volume number 3 and Photoshop installed with a root of /Photodir and is volume number 4, the client would construct a directory for the root of the entire file system containing
File name, Volume number, file number
"Worddir", 3, 0
"Photodir", 4, 0

(The file numbers are both zero here because 0 is the index of the root directory of each volume, and these are the mount points for each volume.)

When new applications are installed, the root of the file system would have to be updated to reflect the newly installed apps.

Filename Parsing
Filename parsing is handled one element at a time, starting at the root of the file system. Parsing one path name element involves reading the parent directory's contents (from the

cache or the app server), searching it for the file matching the next path element's name, and getting the appropriate file id so it can do further lookup.

Volume Versioning... Without File Versioning
We can provide volume versioning and incremental volume updates without versioning each file in the file system. When a new volume is to be provided, we can append any new or changed files as new files in the volume, with new volume IDs that weren't already present. If a directory's contents have changed, then a new version of this directory will be built, with a new file number. This process will proceed from the leaves all the way to the root of the file system, eventually resulting in a new root. The old versions of things would still be available for old clients to access, but clients wishing to access the new version will simply start at the new root, and would thereby get to a consistent picture of the volume. Any file or directory that has not changed from the old version to the new one need not be replicated, and will be referenced by its old file number. (I.e. newly reconstructed directories will contain the old file number for any files that haven't changed.)

If we reserve the first 256 file ids for the root directory, then the version number can be the same as the file number for the root directory.

Note that if we decide that the complexity of this approach is too high, this does not preclude always creating a new volume from scratch for each update.

Constructing File IDs
It is the job of the builder to produce the volume file to file id mapping and to construct all of the directories. Because directories are files identified by file id, this process must begin at the leaves of the volume and proceed to the root.

Note that constructing a new changed volume will consist of finding the diffs between the two volumes and producing some new directories. Changed or newly added files will get new file numbers, leaving the old ones around. Note that any directory that has had any descendents changed must be reconstructed with the new file numbers, and the new directory will get a new file number. This process will proceed to the root of the volume, which will receive a new file number.

Server Failover
All app servers for a particular volume must share the same mapping of file ids to file, so server failover is trivial. There might be a performance impact if the new app server doesn't have the requrested file in memory.

Writing Files into the Application Install Directories
Two approaches have been discussed for the problem of applications that want to write files to their install directories. First, this can be handled wholly inside of the eStream file system. The cache manager could allow writes to files handled by the eFS, but these writes would not be written back to the server. Instead, they would simply be written to

the eFS cache and marked non-purgeable. This approach's primary advantage is that it does not rely on a file spoofer.

The other approach is to use the file spoofer to spoof some accesses to the z: drive. Any open for read/write access would cause the existing file (if any) to be copied to a location on the c: drive, and the file spoofer would then redirect the open to the newly created file. The file spoofer would have to keep track of any file created via this copy-on-write mechanism and redirect all future accesses to the copy. There are some issues to this approach. For example, it is extremely wasteful when files on the z: drive are opened for read/write access but are never actually written. However, it does help reduce the complexity of the eFS cache, and is trivial to implement if we have to do c: to z: file spoofing anyway.

In either case, to support the creation of new files in an application's install directory, it must be possible to modify the contents of directories in the cache.

If we don't use the file spoofing approach, there is the issue of how we support written files when we move to a newer version of a volume. It would probably be necessary to walk the cache and make sure that each written file gets placed in the appropriate place in the new volume version. This is likely to be non-trivial, because we need to have full information about the location of each modified file in the file system tree, and would need to download enough of the new volume directory structure to place these modified files there.

64-Bit File Access?
One question we should answer is whether we will support file sizes greater than 2 GB on the eStream file system. I'm inclined to say that such support isn't a requirement for the 1.0 product, but I also think that the implementation and verification complexity of 64-bit file access on the file system is low enough that we might want to consider building it in anyway.

Simplifications
We could preclude the possibility of an application consisting of more than one volume.

Future Possibilities
Epicon seems to make a big selling point of their technology involving "self-healing" of damaged application files. Such support could be provided by computing checksums on files in the cache. Whether or not we want to support this is an open question. My feeling is that it's something we should leave out of 1.0.

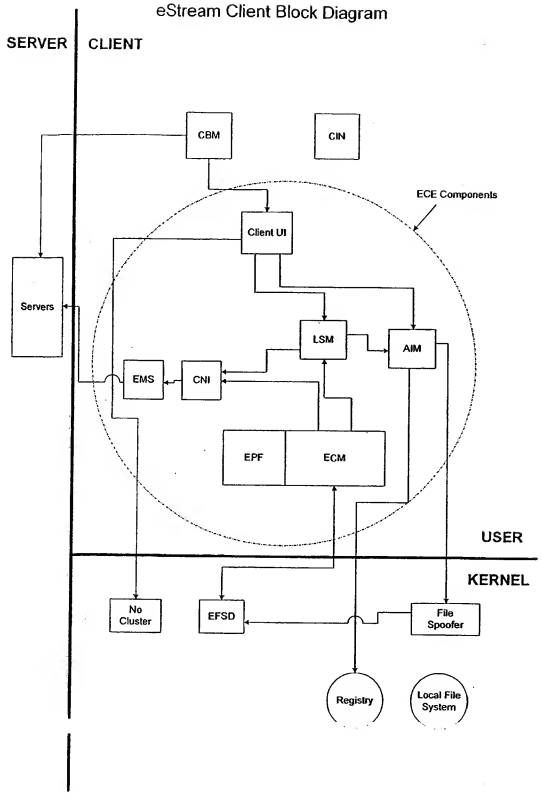
Outstanding Issues
Cache organization has not been addressed.

Finding and downloading the app install block has not been addressed.

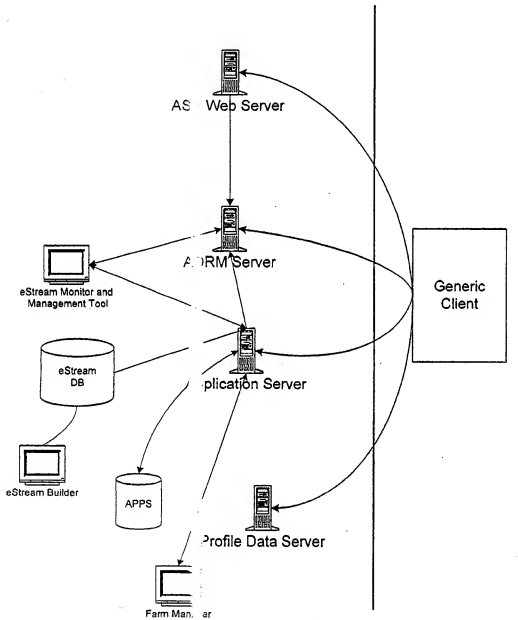
Security in a multiuser system has not been addressed.

eStre .m Builder Block Diagram

???


eStream Client Block Diagram
SERVER
CLIENT
CBM
CIN
ECE Components
Client UI
Servers
LSM
AIM
EMS
CNI
EPF
ECM
USER
KERNEL
No Cluster
EFSD
File Spoofer
Registry
Local File System

eStre ım Server Block Diagram



eStream Requirements from Multiple Perspectives

Ricky Benitez
[redacted]

1.0 OTI Requirements

The following requirements are primarily driven by OTI and its business and operational needs.

1.1 Functionality

- The system can deliver applications to x86-based Windows NT 4, Windows 2000, Windows 95, Windows 98 and Windows Me systems clients in order to capture a substantial size of the existing client market
- The client and server software, software upgrades and application sets can be delivered to customers electronically via the internet (ftp or web) to reduce the costs associated with the delivery of the product
- The client and server software, software upgrades and application sets can be delivered to customers on traditional media (CDs, DVDs) to increase the speed of service at remote sites and to provide the flexibility of servicing isolated networks
- The system allows Omnishift to perform audits, with the permission of the ASP, to ensure that we are being appropriately paid per our license agreement with the ASP

1.2 Localization

- The language used by the server components should be specific to an administrator to facilitate the installation and remote operation of a foreign-run server by OTI personnel

1.3 Usability

- The system must incorporate troubleshooting facilities that allow the support organization to identify and fix issues without engaging the engineering team
- The process of converting applications to eStream sets and testing those eStream sets (certification) must be fairly expeditious (target an average no more than 1 person's effort per application per day)

1.4 Reliability

- The internet-based software, software upgrade and application delivery system must be highly available to support ongoing remote installation and service work

1.5 Performance

- The system must be no less than 50X more scalable than Windows Terminal Server/Citrix Metaframe when running client applications to make it a cost-effective high-volume client application delivery mechanism

1.6 Scalability

- Given sufficient hardware and network resources, the internet-based software, software upgrade and application delivery system must scale to handle any potential number of customers
- The system can be easily expanded to support 64-bit file sizes
- The system provides 128-bit application identifiers
- The system supports applications that are composed of up to 2 billion files

1.7 Security

- It must be difficult enough to steal an application's code and data from the client that software providers are not concerned about using the system as a delivery mechanism
- It must be very difficult to gain unauthorized entry to the internet-based software, software upgrade and application delivery system
- It must be computationally infeasible to steal data or code while it is being distributed across the internet-based software, software upgrade and application delivery system
- It must be computationally infeasible for the internet-based software, software upgrade and application delivery to be coerced by a third-party into delivering a Trojan
- The internet-based software, software upgrade and application delivery must log all accesses so that an appropriate security audit can be periodically performed

1.8 Portability

- The server components must be portable across a wide variety of platforms and operating systems, including but not limited to: Windows NT 4, Windows 2000, Solaris UltraSPARC, HP-UX, and Linux

1.9 Maintainability

- The entire system must be component-wise upgradeable without requiring an interruption of service by any of our customers
- The server components provide a web-based monitor and administrative console that can be used to diagnose and maintain an ASP site
- The client components provide an interface that can be used to diagnose client problems and provide customer support
- All application upgrades are backwards compatible